AFRL-IF-RS-TR-2006-224
Final Technical Report
July 2006

MATTER: MODULAR ADAPTIVE TECHNOLOGY TARGETING EFFICIENT REASONING

SRI International

Sponsored by
Defense Advanced Research Projects Agency
DARPA Order No. S822

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

STINFO FINAL REPORT

REPORT DOCUMENTATION PAGE	*Form Approved* **OMB No. 0704-0188**

1. REPORT DATE *(DD-MM-YYYY)* JUL 06	**2. REPORT TYPE** Final	**3. DATES COVERED** *(From - To)* Dec 04 – Nov 05

4. TITLE AND SUBTITLE MATTER: MODULAR ADAPTIVE TECHNOLOGY TARGETING EFFICIENT REASONING	**5a. CONTRACT NUMBER** FA8750-05-C-0011
	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER 62301E
6. AUTHOR(S) Tomas Uribe, Charles Lieber	**5d. PROJECT NUMBER** T568
	5e. TASK NUMBER 00
	5f. WORK UNIT NUMBER 01
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) SRI International 333 Ravenswood Avenue Menlo Park California 94025-3493	**8. PERFORMING ORGANIZATION REPORT NUMBER** N/A
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Defense Advanced Research Projects Agency 3701 North Fairfax Drive Arlington Virginia 22203-1714 AFRL/IFTC 525 Brooks Road Rome New York 13441-4505	**10. SPONSOR/MONITOR'S ACRONYM(S)**
	11. SPONSORING/MONITORING AGENCY REPORT NUMBER AFRL-IF-RS-TR-2006-224

12. DISTRIBUTION AVAILABILITY STATEMENT
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED. PA#06-484

13. SUPPLEMENTARY NOTES

14. ABSTRACT
The objective of this effort was to investigate novel computer architectures to support machine learning, based on reconfigurable hardware and nanowire growth. The scope of this effort was to bring revolutionary architectural ideas together with application drivers that embody cognitive processing dimensions such as machine learning, large knowledge bases, information security and integrity, real-world reasoning, sensor integration and realtime embedded systems. Conventional processing architectures are ill-suited to processing the large, sparse, graph data structures necessary to efficiently represent cognitive information and computations. Today's silicon hardware can support a large number of parallel operations and high bandwidth and low latency from small, distributed memories. However, traditional von Neumann architectures employ a single-memory, single-instruction stream model that prevents them from fully exploiting the hardware capabilities. This mismatch presents an opportunity to design new hardware architectures that will provide substantially better performance on graph-intensive information processing tasks, which can perform parallel operations over large data structures. To support these tasks while exploiting the silicon, the MATTER architecture described in this report distributes the data structure over a large number of small, fast memories, and associates active logic with each fragment so that it can perform the necessary operations on its local data. In addition this report describes the exploration into nanowire technology, focusing on the growth of new connections. This is a unique capability of nanowire implementations, which could provide a mechanism for adaptation over time.

15. SUBJECT TERMS
machine learning, reconfigurable hardware, nanowires

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON** Christopher Flynn
a. REPORT U	**b. ABSTRACT** U	**c. THIS PAGE** U	UL	65	**19b. TELEPONE NUMBER (*Include area code)***

Table of Contents

List of Figures

List of Tables

Executive Summary

Conventional processing architectures are ill-suited to processing the large, sparse, graph data structures necessary to efficiently represent cognitive information and computations. Today's silicon hardware can support a large number of parallel operations and high bandwidth and low latency from small, distributed memories. However, traditional von Neumann architectures employ a single-memory, single-instruction stream model that prevents them from fully exploiting the hardware capabilities. This mismatch presents an opportunity to design new hardware architectures that will provide substantially better performance on graph-intensive information processing tasks, which can perform parallel operations over large data structures.

To support these tasks while exploiting the silicon, the MATTER architecture distributes the data structure over a large number of small, fast memories, and associates active logic with each fragment so that it can perform the necessary operations on its local data. The small memories are connected by an efficient, high-bandwidth network, so that we can quickly bring together separate pieces of the data structure needed to perform calculations.

FPGAs provide hardware technology that can be used to instantiate the MATTER architecture today. While small graphs can be directly implemented spatially in FPGAs, the size of graphs that can be realized with a modest number of FPGAs is extremely limited. Consequently, we introduce a new concurrent system architecture for sparse graph-processing algorithms. The system architecture provides a high-level way to capture a large range of graph-processing tasks abstracted from the detailed hardware implementation. We can efficiently map tasks in this system architecture to collections of FPGAs with embedded memories, allowing performance to scale with the number of FPGAs used to solve the problem.

As a sample graph-based cognitive application, we consider the ConceptNet Knowledge Base. On typical queries, the MATTER implementation yields an order of magnitude speedup per FPGA compared to a state-of-the-art Pentium processor—provided that we have a sufficient number of FPGAs to fit the task.

Future technology will help us better realize the large capacity that MATTER requires. Nanowire technology, in particular, has the potential of shrinking large-capacity MATTER systems down to a single chip (perhaps with an even larger memory ratio). Thus, part of this project explored the basic science of nanowire technology, focusing on the growth of new connections. This is a unique capability of nanowire implementations, which can provide a mechanism for adaptation over time. While full deployment of in-field nanowire growth or assembly still requires much research and development, the MATTER architecture provides a concrete application driver for this new capability as it makes the transition from science to technology.

Chapter 1

Introduction

This document is the final report for the MATTER project. MATTER (Modular Adaptive Technology Targeting Efficient Reasoning) is a collaborative effort between SRI International, Caltech, MIT, and Harvard University, with the goal of designing next-generation hardware to support cognitive applications.

Outline: Chapter 1 presents an overview of the MATTER graph machine. Chapter 2 describes, in more detail, a system architecture based on the MATTER ideas. Chapter 3 briefly describes the Dishoom FPGA platform, a prototype hardware implementation of the MATTER architecture. Chapter 4 presents the project's results in nanowire chemistry, including nanowire growth and reconfiguration. Chapter 5 summarizes the activities under the contract.

Appendixes A and B present detailed analysis and performance estimates for our target cognitive application, ConceptNet.

1.1 Overview of the MATTER Graph Machine

Conventional processing architectures, such as commodity desktop (*e.g.* Pentium 4), server (*e.g.* Itanium), and supercomputer (*e.g.* Cray) processors, are ill-suited to processing the large, sparse, graph data structures necessary to efficiently represent cognitive information and computations, such as semantic and Bayesian nets, knowledge bases, and graph search.

Today's silicon hardware capabilities can support a large number of parallel operations and high bandwidth and low latency from small, distributed memories. However, traditional von Neumann architectures employ a single-memory, single-instruction stream model that prevents them from fully exploiting the hardware capabilities. Most of the processing area goes to supporting the single-memory, single-instruction abstraction rather than supporting the actual computation. Memory bandwidth is artificially bottlenecked through the single-memory abstraction, preventing the efficient use of the memory that the hardware can provide. Even within the memory bandwidth that traditional architectures do provide, they can approach peak bandwidth only when the data structures are regular (large, contiguous blocks of memory as used in

	Conventional	MATTER
Specialized Datapath (cycles)	1000	1–10
Small Local Memories (cycles)	100	1
Lightweight, Low-Latency Network (cycles)	10^3–10^5	10
Parallel Operations per Chip	1	100

Table 1.1: Sources of Speedup in the MATTER Graph Machine

arrays of primitive data types).

This mismatch between the capabilities of the silicon and the conventional architectural model presents an opportunity to design new hardware architectures that will provide substantially better performance on graph-intensive information processing tasks, which can perform parallel operations over large data structures. To support these tasks while exploiting the silicon, the MATTER hardware architecture distributes the data structure state over a large number of small, fast memories and associates active logic with each piece of the data store so that it can perform the necessary operations on its local data. The small memories are connected by an efficient, high-bandwidth network, so that we can quickly bring together diverse pieces of the data structure to perform calculations.

1.1.1 Optimization Prospects

A graph machine can provide orders of magnitude higher performance than conventional alternatives on graph-intensive applications. The optimization prospects, listed in Table 1.1, are easy to understand:

1. By specializing the datapath for common graph operations, such as marking a node or an edge, each graph operation can be lightweight, taking only 1 to 10 cycles.

2. By using small, distributed memories, each graph operation completes in a few cycles, rather than taking hundreds of cycles to fetch data from a large, distant memory.

3. Using a low-latency, efficient network, data can be routed between distributed graph nodes in 10 to 100 cycles with minimum protocol and switching overhead.

4. Exploiting many distributed memories and distributed processing, many graph operations may occur in parallel.

Lightweight operations and local memories make operations that can take thousands of cycles on a conventional processor operate in approximately 10 cycles. Low-latency interconnect takes connection links that are normally 1000 to 100,000 cycles down to approximately 100 cycles. Using a large number of small, simple logic/memory blocks allows hundreds to thousands of operations to occur in parallel on a modest silicon die, whereas conventional processor architectures do well to complete 1 to 10 operations per cycle.

1.1.2 Graph Model

The abstract model for the user is a graph of object nodes. Conceptually, each node is its own locus of execution (its own thread) and all nodes operate concurrently. Each node receives messages (method calls) with data and can, in turn, send messages to (invoke methods on) any of the other graph nodes to which it is connected in response to the incoming message. Messages/methods may modify the object data. Each object has a set of defined operations. Data can be accessed only through the defined object methods.

1.1.3 Folding onto Memory

In the extreme, each object could have its own physical resources. The hardware-assisted router [DHW02] and the spatial annealer [WD03] were designed to this extreme. In general, however, we will most likely group a set of graph nodes into a single memory and share a single graph processing node. This folding is an implementation decision and should not change the semantics of the graph operations. An important question for a given technology will be the right virtualization factor—that is, the appropriate number of nodes to share a physical graph operator.

A common optimization would be to place graph nodes of a single type on a single, physical graph processing node. In this way the logic executing on (or implemented by) the graph processing node can be tightly specialized for the single type of data object in its memory. This is, however, an optimization, and the graph node could be generalized where appropriate.

1.1.4 Graph Node Implementation Microarchitectures

There are many different microarchitectures we might use to implement a graph processing node. For example,

- Use a simple, generic microprocessor and compile the graph operations for a particular graph node type to it.
- Design a specialized processor optimized for generic graph operations and compile the graph operations for a particular graph node type to it.
- Provide a fixed amount of reconfigurable hardware for the graph node processing and compile the graph operations for a particular graph node to it.
- Define a standard format for graph nodes (analogous to IEEE-754 as a standard format for floating-point representation) and a standard set of common graph operations, and build a specialized hardware datapath for handling.

Abstractly, the architectural goal would be to admit these implementations (although the last would admit a different architectural view than the other three). Quantitatively, we can then evaluate which is most suitable for a given technology. We might even mix and match within a system or implementation. For example, on an FPGA implementation, we might start using a generic, soft processor for a graph node (easy compilation target). We could transition to a specialized soft processor by extracting special graph operations for the particular object. We could later compile the graph node operations directly to a VHDL or LUT-level implementation.

1.1.5 Interconnect Microarchitecture

The model for graph nodes is that they are directly connected to their neighbors. However, there are many ways we could implement the connections:

- static (FPGA-like) configurable network
- circuit-switched connections
- packet-switched connections
- bus or hierarchy of buses
- shared memory

Different network structures will be appropriate based on the common graph usage pattern (Section 1.1.6) and the folding (Section 1.1.3).

1.1.6 Graph Usage Patterns

Different applications have very different usage patterns for graphs. It will be useful to identify the patterns, optimize for them, and, perhaps, allow higher-level programming to provide hints or directives as to which case is most appropriate.

- **static graph** – the graph is constructed once and then used repeatedly. The graph structure does not change during the bulk of operation. Values at the graph nodes may change. Performing queries and evidence updates on a static knowledge corpus might have this characteristic. Placement and routing on fixed architectures has this characteristic.

- **quasi-static graph** – the graph is incrementally changed, but changes are relatively infrequent compared to the other graph operations. A large knowledge base (10^6+ nodes) that has only a few (10s) of changes per hour would certainly fit this model.

- **dynamic graph** – the graph is generated and changed regularly. A graph generation operation may be as complex as, or more complex than, any operations run on the graph once created. In another case, a large fraction of the graph links change every few cycles.

Of course, it is an oversimplification to say that the application will exhibit a single one of these patterns. It may be more accurate to say that each data structure in an application may exhibit one of these patterns. For example, an application might have a static knowledge base, and then abstract a series of instance-specific Bayesian networks from it.

1.1.7 Results: Speeding Up Cognitive Applications

In Appendix A and Appendix B, we analyze the propagation algorithm used by ConceptNet [LS04], a commonsense reasoning tool from MIT. ConceptNet's semantic network contains 300,000 nodes and 1,600,000 edges, with 25 distinct link types. Starting with a set of nodes of interest, weights propagate through the graph, discounted by the types of edges, until a threshold is reached. A final phase ranks the nodes (concepts) based on the weights they have received. This style of propagation algorithm is similar, for instance, to the Probabilistic Relational Neighbor (pRN) algorithm [MP03] used in the CADRE system [WF05].

ConceptNet includes challenges for MATTER mapping, such as a small number of very highly connected nodes, labeled edges, and irregular communication and activity patterns. In spite of these challenges, our results show that a speedup of three orders of magnitude is possible, when compared with a high-end commercial sequential processor (e.g., Pentium 4).

Other relevant work, where we have analyzed proto-graph-machine field-programmable gate array (FPGA) applications, includes

- Routing (path search, effectively a multicommodity flow optimization) [DHW02, HWD03]
- Placement (simulated annealing, starting point for partitioning, clustering) [WD03]
- Sparse Matrix-Vector Multiplication (example of sparse connectivity and demonstration of core routine for numerical processing) [dD05]

In routing and placement, the FPGA implementation can achieve speedups as high as three orders of magnitude over the traditional implementations.

The following chapter describes the MATTER architecture in more detail.

1.2 Nanowires and MATTER

FPGAs are one way to instantiate the MATTER architecture, using current and readily available technology. In the longer term, nanowire technology offers a promising avenue for implementation of MATTER-like architectures in the future.

The MATTER architecture provides a good match to the large capacity that nanowire hardware enables. In particular, it is clear that conventional architectures do not scale to even exploit their own available hardware well. Thus, there is a need for architectures that can bring very large amounts of hardware capacity to bear on a problem.

Conversely, we can look at the nanowire implementation as an answer to the question of how to realize the large capacity that MATTER requires, integrating memory and compute devices. Certainly, one could argue that our current-technology, FPGA-based prototypes are excessively large, requiring hundreds of chips. Looking ahead, those kinds of systems shrink down to a single chip (perhaps with an even larger memory ratio) when realized using the nanowire architectures.

The nanowire chemistry component of this project, described in Chapter 4, explored the prospect of adding nanowires as a unique capability enabled by the nanowire implementation. This allows us to consider features such as self-repair (which is increasingly important at these small feature sizes) and adaptation (to application, environment, and dataset). Particularly for cognitive applications, we expect the machine to need to learn and adapt over time. This in-field addition of nanowires provides another mechanism to tailor the machine to the application.

Full deployment of in-field nanowire growth or assembly still requires much research and development. However, the MATTER architecture provides a concrete application driver for this new capability as it makes the transition from science to technology.

1.2.1 Adaptive Growth

Controlled nanowire growth, such as described in Chapter 4, also has the potential of allowing circuit specialization over time. New connections can be created and destroyed depending on the characteristics of the task being performed by the circuit.

The ability to grow new wires (and remove old ones) adds new adaptation possibilities at run-time. If we map the data onto the hardware, we can adjust to small incremental changes on the data. Consider a cognitive application such as ConceptNet (which we analyze further in the following chapters). As the knowledge base changes, we would like to change the weights and connections in the hardware encoding of the graph. A fixed wiring scheme, that includes all connections a priori, would require orders of magnitude more memory and wiring than the graph itself, increasing the bandwidth cost that is already dominant in the application. If we want to exploit sparseness in the data, dynamic reconfiguration is essential.

When re-organizing the data on the existing hardware is not enough, new connections can make new configurations possible to alleviate bandwidth problems.

Online Partial Evaluation: Just as partial evaluation is analogous to the process of compiling some aspects of the problem formulation onto the hardware, reconfigurable hardware can be used for *online* partial evaluation, which has been more commonly investigated in the case of software. Often, there are characteristics of the input data, context and environment that can only be detected at runtime, but which are also useful for optimization.

If we implement a nano-MATTER architecture, and the nanowire chemistry gives us exact control of the design, we may be able to optimize the hardware with respect to the operating environment as detected at runtime, through periodical use of “sleep” intervals, during which the system reconfigures itself.

Chapter 2

A System Architecture for Sparse-Graph Algorithms

Summary: This chapter describes GraphStep, a system architecture that embodies the main ideas in the MATTER Graph Machine. Many important applications are organized around long-lived, irregular sparse graphs (*e.g.*, data and knowledge base application, CAD optimization, numerical problems, simulations). The graph structures are large and the applications need regular access to a large, data-dependent portion of the graph for each operation (*e.g.*, the algorithm may need to walk the graph, visiting all nodes, or the algorithm may need to propagate changes through many nodes in the graph). On conventional microprocessors the graph structures exceed on-chip cache capacities, making main-memory bandwidth and latency the key performance limiters. To avoid the "memory wall," we introduce a concurrent system architecture for sparse graph algorithms that places graph nodes in small distributed memories paired with specialized graph processing nodes interconnected by a lightweight network. This gives us a scalable way to capture and map these applications so that they can exploit the high-bandwidth and low-latency capabilities of embedded memories (*e.g.*, FPGA Block RAMs). On typical spreading-activation queries on the ConceptNet Knowledge Base, this translates into an order of magnitude speedup per FPGA compared to a state-of-the-art Pentium processor.

2.1 Comparison Notes

The primary comparison is an order-of-magnitude speedup *per FPGA*, assuming that we have sufficient FPGAs to perform the task (more on this below). This is a deliberately *conservative* comparison to standard processors. The best that conventional processors could do is to get a linear speedup with the number of processors, requiring no additional resources for interconnect. (We have charged the FPGA architecture for using FPGAs for interconnect.)

In practice, it is highly unlikely a conventional multi-processor implementation would scale linearly with the number of processors, given the difficulty in parallelizing the application in a sequential environment. In fact, there is evidence that Pentiums (or ASCI machines) perform abysmally on these sparse graph operations. So, in practice, if we were to make a direct comparison between the two, the conventional processor would fare even worse.

Separately, a common objection is that applications are serial-bottlenecked, so one cannot do better than building a fast serial processor. Our results show, for an important class of problems, that the parallelism is there and can be exploited. Therefore, for these problems there are much better approaches than just building the fastest sequential-processor possible.

The above comparison assumes that we are willing to dedicate a sufficient number of FPGA's to the task. (The analogue of this, in the single-processor case, is that we buy enough RAM.) If the goal is to minimize the area, and one does not care about performance, then the processor may still be the right way to go. However, we're showing that we can exploit the silicon capacity that is available now (and will be even more available in the future) to do much better. This, even though FPGA's are not the perfect memory vs. compute balance point for MATTER. Nonetheless, we show that they are good enough to deliver a significant performance and capability advantage. With further research, we may zero in on a better balance point, which will show even greater advantage.

2.2 Introduction

We have long noted the fact that spatial hardware organizations, such as FPGAs and other reconfigurable architectures, offer computational density superior to that of more conventional, temporal hardware organizations [DeH00, DeH96]. Conferences such as FCCM (Field-Programmable Custom Computing Machines), where a version of this chapter will appear [dKM$^+$06], have showcased numerous compute-intensive applications where FPGAs deliver performance that is orders of magnitude superior to that of processor-based systems. Further, we are beginning to see high-level system architectures for capturing these compute-intensive applications in scalable manners (*e.g.*, SCORE [CCH$^+$00], and cellular automata models [DAd$^+$04, MCMB93, STO03, KMH01, Mar97]).

Nonetheless, many problems are limited by memory speed rather than compute speed. As processing speeds continue to increase faster than memory speeds, the effect is exacerbated, leaving many applications limited by memory performance rather than compute performance [WM95, McK04].

Spatial organization of computations turns many memory operations into interconnect [DeH96]. Nonetheless, it often remains infeasible to implement tasks with large data sets in a fully spatial manner (*e.g.*, [dD05]), leaving a need to use memories for virtualization. To address this, modern FPGAs integrate increasingly larger quantities of on-chip memory. The aggregate memory bandwidth accessible from the collection of small, distributed memories is two orders of magnitude larger than the memory bandwidth available on processors (Section 2.3). This presents a new opportunity for FPGAs to offer superior performance to microprocessors on data-intensive applications.

Family	Pentium-4	Virtex-2	Virtex-4	Stratix-2
Chip	Pentium-4 550	XC2V6000	XC4VLX200-12	EP2S180
Technology	90 nm	150 nm	90 nm	90nm
Memory Clock	3.4 GHz	260 MHz	500 MHz	475 MHz
On-chip Memory BW	0.2 Tb/s	1.2 Tb/s	5.4 Tb/s	12 Tb/s
from	L1 D-Cache	144 BRAMs	336 BRAMs	768 M4Ks
On-chip Memory Capacity				
at speed quoted	16 KB	288 KB	688 KB	192 KB
total	1 MB	0.29 MB	0.69 MB	1.1 MB
Off-chip Memory BW	51 Gb/s	77 Gb/s	110 Gb/s	77 Gb/s
Reference	[Int05]	[Xil03]	[Xil05]	[Alt05]

Table 2.1: Raw Memory Bandwidth Available from FPGAs and Processors

Algorithms that represent data with sparse graphs are a large class of these data-intensive applications. While small graphs can be directly implemented spatially in FPGAs (*e.g.*, [BFA96, MHH02]), the size of graphs that can be realized with a modest number of FPGAs is extremely limited. Consequently, we introduce a new concurrent system architecture for sparse graph-processing algorithms. The system architecture provides a high-level way to capture a large range of graph-processing tasks abstracted from the detailed hardware implementation. We can efficiently map tasks in this system architecture to collections of FPGAs with embedded memories, allowing performance to scale with the number of FPGAs used to solve the problem. The new system architecture is complementary to compute-intensive system architectures like SCORE, providing a natural way to capture these data-intensive applications.

The novel contributions of this work include:

1. Highlighting the raw, memory-bound performance potential of FPGA hardware
2. Introduction of data-centric system architecture for sparse-graph applications
3. Mapping of new system architecture to FPGAs with a collection of small distributed on-chip memories
4. Demonstration of performance benefit on a sample application
5. Identification of class of applications that conceivably benefit from the performance potential using this system architecture

2.3 Raw Memory Performance

Table 2.1 summarizes the raw, aggregate memory bandwidth available on processors and FPGAs to both on- and off-chip memory. In each case, this is computed in the most simplistic and direct way. For the processors, on-chip bandwidth is the bandwidth available from L1 memory. For the FPGAs, on-chip bandwidth assumes that specified RAMs (Block RAMs, M4Ks) operate concurrently at their dual-port operating speed (given by the memory clock speed) and width transfering data on both ports. For the FPGA off-chip bandwidth, we assume that the off-chip pins are dedicated SDRAM interfaces (twelve 32b SDRAM interfaces operating at 200 MHz for Virtex-2, twelve 32b SDRAM interfaces operating at 300MHz for Virtex 4, eight 16b SDRAM interfaces operating at 300 MHz on two edges for Stratix 2).

We can make several important observations from this data:

- A single FPGA can offer higher on-chip memory bandwidth than the most advanced microprocessors—one to two orders of magnitude at comparable technology generations.
- For the FPGA, the on-chip bandwidth is one to two orders of magnitude higher than off-chip bandwidth; further, we expect on-chip capacities and hence potential bandwidth to increase more rapidly than off-chip bandwidth, widening the on-chip vs. off-chip bandwidth gap.
- Assuming we can exploit the parallelism, we can scale bandwidth in large systems by tiling FPGAs; similarly, vendors scale the on-chip bandwidth along with compute capacity by scaling the number of independent, on-chip memory banks.

These are, of course, peak memory numbers. Neither architecture is likely to achieve them. Processors can seldom run entirely from L1 memory, and practical caching schemes fail to exploit the potential bandwidth available (*e.g.* [HS95]). Nonetheless, these observations do point to real performance ceilings and raw potential that we may be able to exploit.

Further, traditional ways of organizing computations result in very significant deviations from these peaks when the dataset is large. That is, traditional processor applications will fetch data and stall execution until the data is returned (allowing multiple, outstanding memory references helps, but does not completely compensate for this strategy). Consequently, when the dataset is large and cannot be contained in the on-chip memory, bandwidth becomes limited by the off-chip access latency rather than the on- or off-chip bandwidth. Consequently, access bandwidth may easily drop another order of magnitude.

2.4 Idea

If we could arrange for all of our data to reside in distributed on-chip memory (*e.g.*, FPGA Block RAMs), and arrange to perform parallel operations and hence parallel access to the data, we could exploit this raw potential (Section 2.3) and achieve orders of magnitude improvement in net memory bandwidth and hence performance on data-centric processing tasks. To handle large tasks, we assemble multiple-FPGA collections to contain the data. This gives us two additional wins:

1. We scale bandwidth and processing with the dataset.

2. We keep all data within a constant (small) latency of the active processing.

Of course, we get less memory capacity per die (per λ^2 or per cm^2 of silicon) using memory in an FPGA than we get using off-chip DRAMs. This is a deliberate tradeoff we make to get higher performance on these tasks. If performance is limiting the application, then this gives us a way to trade area to obtain higher performance.

We can also engineer FPGAs with a different memory/logic balance or with embedded DRAMs (*e.g.*, [MAS$^+$97, PJA$^+$99]) that would provide an architectural point between these extremes. These architectures might trade only a factor of 2 to 3 in net memory density for orders of magnitude improvement in usable memory bandwidth.

2.5 Graph Applications

Many graph processing applications are naturally represented by sparse graph data structures and can exploit the opportunity identified in the previous section. In these problems:

- The graph is sparse and irregular, meaning nodes have a bounded [$O(1)$] number of edges, but are not necessarily connected in nearest-neighbor fashion in any number of dimensions. Because of the irregular connectivity and data access, it is not possible to localize processing to a small subset of the graph; *i.e.*, traditional *spatial locality* exploited in cache-line blocking and virtual memory pages is not adequate to hide the long delay to off-chip memory on processors.
- Algorithms require that the whole graph (or large fractions of the graph) be traversed as part of an iteration.
- Algorithms admit to parallelism across the graph.

To be concrete, consider the following kernels and applications:

- **Iterative Sparse Matrix-Vector Multiply** – Here we must complete each sparse matrix-vector multiply (SMVM) before starting the next, and each SMVM requires that we access all the sparse-matrix coefficients. Each entry in the vector result is independent and can be computed in parallel [dD05].
- **Sparse Neural-Network Evaluation** – This can essentially be the same problem as SMVM above.
- **Shortest Path** – A traditional (*e.g.*, Bellman-Ford [CLR90]) shortest path computation requires that every node update its delay on every cycle. The serialization goes only as the depth (diameter) of the graph, which is typically small compared to the size of the graph for high-speed circuit graphs.
- **Routing** – Routing (*e.g.*, FPGA routing such as Pathfinder [ME95]) is based on a series of shortest path searches. For nets that cross the entire device, the shortest path search can potentially touch the majority of routing resources in the circuit. When nets are highly localized, it may be possible to perform multiple route searches on different portions of the device in parallel. We already have evidence that this parallelism can lead to substantial speedups in routing [DHW02, HWD03, Hua04].
- **Timing Calculations** – Simple timing analyses (ASAP and ALAP calculations) also perform whole graph traversals in order to update delays and slack.

- **Placement** – Node placement can move a large number of nodes, potentially all of them, and update their costs in parallel [WD03, Wri03].
- **Associative Search** – In some applications, we need to check every graph node for some property.
- **Transitive Closure** – Transitive closure is a reachability search that can be seen as a simplified version of the shortest path problem.
- **Marker Passing** – Many knowledge-base queries, inferences, and classification tasks can be supported by algorithms that propagate binary data along neighboring links and perform local and global binary state operations [Fah79, KM93].

In general, any application that needs to walk the entire graph will fit the properties noted above, particularly when the operations at each node can be cast as one of the following:

- perform operation at a node (data parallel)
- accumulate information from nodes (associative reduce)
- propagate information to neighboring nodes

2.6 GraphStep System Architecture

To exploit the idea introduced above, we develop the GraphStep concurrent system architecture. We call this a concurrent system architecture in the spirit of "Software Architectures" [SG96], and, in fact, the GraphStep architecture is closely related to an Object-Oriented or Repository software architecture. As a concurent system architecture, GraphStep gives a gross organization for conceiving the task and managing the parallelism in the task.

2.6.1 System Architecture Description

In the GraphStep architecture, the computation is organized as a graph of nodes connected by edges.

Nodes: Each node is an object or actor [HBS73]. That is, it has:

- local state, typically in typed data fields
- edges to other graph node objects along which it can send messages or method invocations
- a set of methods through which the object data is accessed and modified

It can be useful to think of each object as having its own locus (thread) of control and acting concurrently with all other objects. The program counter is part of its local state. As explained below, the objects synchronize in "steps", so it is alternately possible to simply think of the objects being invoked in a data-parallel, concurrent manner and performing operations that depend on their state.

Methods: In strict, object-oriented fashion, the object can be accessed only through its methods. Most methods are invoked through messages from edges (connected objects), although methods can also be self-invoked or invoked globally (typical in broadcast operations). Methods are of bounded length and atomic. Self-invoked methods may be used to perform recursive operations on a single node. In response to a method invocation, an object may change its state and send a message (*i.e.*, method invocation) along each of its edges or may produce a message into a global reduce operation.

Graph Operation: The graph evaluates as a series of synchronized steps. The evaluation model is a Receive-Update-Send sequence:

1. Graph nodes receive input messages.
2. Graph nodes wait for an activation signal to proceed.
3. Graph nodes perform an update operation.
4. Graph nodes send output messages.

This evaluation sequence is the basis of semantic correctness and scaling. Graph node operations appear concurrent in that all nodes perform their update and exchange messages between synchronization events regardless of how they are sequentialized onto physical processing engines. Deterministic computation is guaranteed by forcing a step's set of messages to be received before performing each update. The GraphStep name was selected to emphasize this step-by-step operation.

Global Operations: A central controller can perform global broadcast and reduce operations on the graph or an activated subset of the nodes in the graph. The broadcast operations are effectively a designated method invocation on every node.

2.6.2 Relation to Other Concurrent System Architectures

The GraphStep architecture can be seen as a stylized restriction of the Bulk-Synchronous Parallel (BSP) model [Val90]. Like BSP, its semantics are based on a series of steps synchronized across the entire machine. The GraphStep architecture is more stylized in that it restricts the computational tasks to method updates on an object graph and emphasizes communication along object links, whereas BSP takes no stand on how communication occurs.

GraphStep can also be seen as a Data Parallel model in that operations are performed on a set of concurrent objects. The operations are not necessarily homogeneous actions applied to data because:

- Nodes may be of different object types.
- Nodes often depend on methods invoked, which may differ within a single operational step.

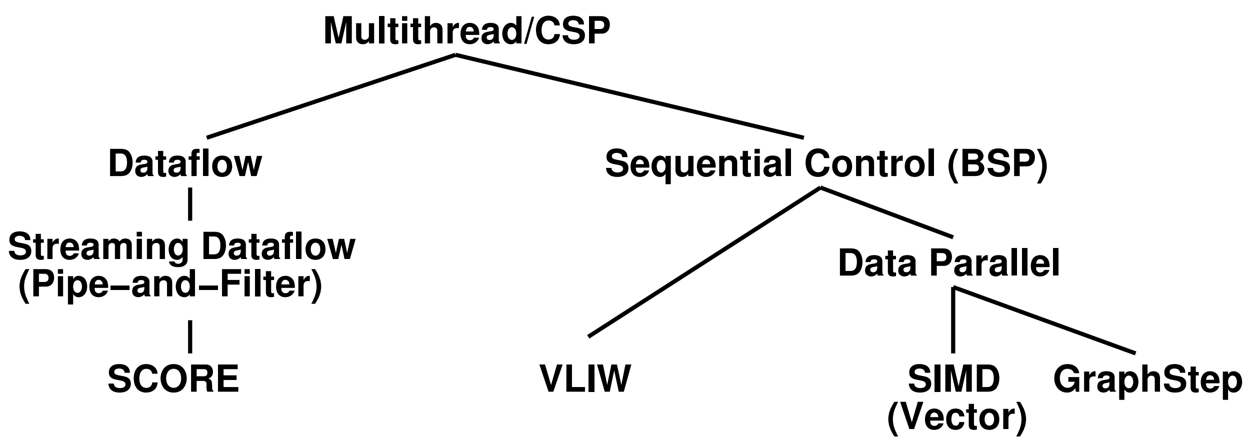


Figure 2.1: Portion of Concurrent System Architecture Taxonomy Placing GraphStep

The SCORE architecture [CCH+00] also organizes computation as a graph of nodes. However, there is a fundamental difference between the semantics of the SCORE model and the GraphStep model in that SCORE is based on dataflow semantics, while GraphStep is based on lock-step sequential semantics. That is, SCORE nodes (operators or “filter” using the “pipe-and-filter” terms) synchronize only on the presence of data on their inputs, allowing some nodes to run ahead of other nodes as long as they have present data. In GraphStep all nodes are allowed to evaluate each step. In SCORE, a computation may wait for a set of inputs to occur, whereas in GraphStep, the node processes all the edges that have arrived on a cycle, even when this is only a subset of the potential inputs. One consequence of the dataflow semantics is that SCORE allows unbounded FIFOs on the edges (streams, pipes) between nodes, whereas GraphStep demands that all messages be delivered and consumed synchronously.

Philosophically, GraphStep is a data-centric concurrent system architecture and consequently takes a very different stand on how computation progresses than either SCORE or traditional, multithreaded computations. In GraphStep, the graph is the fixed point and computations are sent to the data. In SCORE, the graph is the computation and data is streamed through the graph. In a traditional processor organization, the computation runs on a processor and data is fetched from memory (possibly remote memory) to the processor in order for computation to proceed. Consequently, multithreaded, processor-oriented computations always involve a round-trip message pair to acquire data. Without careful latency-hiding hardware (*e.g.*, [AI87, SCB+98]), the round-trip latency for data fetches can end up limiting exploitable data bandwidth and computational throughput. In contrast, GraphStep operations have a Continuation Passing Style (CPS) (*e.g.*, [AJ89]) with execution always moving to the data.

Figure 2.1 shows a piece of the concurrent system architecture taxonomy, illustrating how GraphStep is related to the other architectures discussed in this section.

2.6.3 Possible Realizations

The concurrent system architecture defines the way the computation should be organized and expressed, as well as its semantics. While preserving the semantics, the architecture admits to a wide range of implementations. For example:

- **Fully Spatial** – The entire graph can be implemented spatially, with each node getting its own processing engine and with dedicated links between graph nodes. The graph may be configured on top of one or more FPGAs (*e.g.*, [BFA96, MHH02, HWD03]).
- **Sequential Processor** – The entire graph could be processed by a single processor that picks up each node and executes it in turn. In this case, during data propagation steps, when no global operations are performed, the implementation may keep an active node set [Hil85] so it can avoid visiting nodes that have received no messages during the previous GraphStep `send` operation.
- **Multiprocessor** – The graph nodes can be distributed among the nodes of a multiprocessor. Each processor is responsible for evaluating its nodes in sequence. This could even be realized using multiprocessor chips with local memory such as MIT's RAW [WTS$^+$97] or IBM's Cell [PAB$^+$06]. Processor-in-memory (PIM) message-passing nodes would also let us exploit a close coupling of on-chip memory and data (*e.g.*, [LRSS84, DFK$^+$92, SMK$^+$96]).
- **Specialized Graph Processor** – It may be useful to build specialized processors designed to handle the typical operations involved in handling graph node messages. This could include integrated message handling (*e.g.*, [HJ92, LDK$^+$98]).
- **Reconfigurable with Embedded Memories** – the graph nodes can be distributed among specialized graph processing nodes configured on top of an FPGA with the nodes associated with each graph processing node stored in on-chip, embedded memories (*e.g.* Block RAMs; see Section 2.7.4).
- **Object-Specialized Graph Processing Engines** – when implementing the nodes on top of an FPGA, we can potentially assign graph nodes to processing nodes by object type and specialize each processing node to handle a single type of node object.

In practice, the fully spatial case is unlikely to be ideal when supporting graphs with thousands of nodes. In particular, the GraphStep model demands that we complete communication between phases. That means we must wait for the worst-case communication latency between nodes in the graph. If this latency is large (*e.g.*, hundreds of cycles) compared to the processing of a single message or node update (*e.g.*, 1 to 10 cycles), then a fully spatial implementation will spend all of its time waiting for messages to be routed. Consequently, sharing a processing node among a modest number of graph nodes will better balance out the computation and communication latency. Effectively, this allows us to use substantially less hardware without increasing execution time; since the worst-case communication distance shrinks with the size of the physical hardware, up to a point, this may yield a net reduction in the time required for each GraphStep.

Ultimately, node serialization will dominate communication latency and further serialization comes at the expense of slower computation.

2.7 Example: ConceptNet

As a concrete example, we consider an FPGA implementation of spreading activation on the ConceptNet Knowledge Base [LS04] and compare this to a C-coded, sequential Pentium implementation.

2.7.1 Knowledge Base

ConceptNet is a knowledge base for common-sense reasoning compiled from a Web-based, collaborative effort to collect common-sense knowledge [LS04]. Nodes in the ConceptNet knowledge base are nouns and verb-noun pairs (*e.g.*, "run marathon"). Edges are distinguished by type to denote specific semantic relationships (*e.g.*, "effect of", "used for"). The knowledge base is used in natural language processing and commonsense reasoning tasks. Specific applications to date have included identifying contextual neighborhoods, topic gisting, analogy generation, predictions from sensor data, semantic prediction (projections), disambiguation, and affect sensing.

A "small" version of the ConceptNet knowledge base contains more than 14K nodes and 27K edges. The default ConceptNet knowledge base contains 220K nodes 550K edges. There are 25 types of semantic relationships.

2.7.2 Spreading Activation

A key operation on the ConceptNet knowledge base is spreading activation. In spreading activation, an initial set of graph nodes are activated; these may be keywords or portions of a natural language text. Based on the application, each edge is given a weight coefficient based on its type. Starting with an activation potential of one (1.0) for the initial set of nodes, activities are propagated through the network, stimulating related concepts. After a series of propagation steps, each node in the network will have an updated activity factor. Typically, nodes with the highest activity factors are then identified as being most relevant to the initial query. The spreading activation calculation is similar to neural-network simulation, the difference being the source of the links and weights, and the fact that the link weights vary based on the application in which ConceptNet is used, as well as the specific query being performed.

Figure 2.2 describes the spreading activation calculation. For actual implementation, this can be optimized while achieving the same semantics. Sequential implementations can take care to visit only nodes that receive at least one input message in a step. Since the update operation is associative, an implementation can directly sum the message into *step-activity* without waiting for the update phase; this avoids the need to make a full pass over the inputs during the update phase and avoids the need for space to store the full set of input activities in a step. To avoid buffering all the incoming messages, send and receive phases can be overlapped.

```
AUPDATE(v1,v2)
    tmax = max(v1,v2)
    tmin = min(v1,v2)
    return(tmax+(1-tmax)×tmin)
```

```
SPREADINGACTIVATION
//start with activities of non-initial nodes set to zero
foreach step
    foreach graph node g
        // receive
        foreach incoming message m
            g.edges[m.edge].activity←m.activity
        wait for step synchronization
        // update
        g.step-activity←0
        foreach input edge e to g
            g.step-activity ← AUpdate(g.step-activity,
                                        e.activity)
        g.node-activity ← AUpdate(g.node-activity,
                                        e.activity)
        // send
        foreach output edge e from g
            if (g.step-activity>THRESHOLD)
                send to e.sink with
                    activity=g.step-activity×g.discount
                            ×weight[e.type]
        // reset
        foreach input edge e to g
            e.activity←0
```

Figure 2.2: Basic Computation for Spreading Activation Calculation

2.7.3 Sequential Implementation

For baseline comparison, we implemented a streamlined version of spreading activation in C to run on standard microprocessors. The default ConceptNet graph requires more than 30 MB to represent and, consequently, will not fit in the 1 MB on-chip cache on Pentium processors. Even the smallest ConceptNet graph requires 1.5 MB to represent.

To optimize the sequential implementation, we use an active graph node queue so that we need to visit only the nodes that have new activity on each graph step. We also use an efficient radix sort data structure (similar to the one used in [FM82]) so we can extract the highest-activity nodes without walking the entire graph or paying $O(N\log(N))$ to perform the sort. Both insertion into the activity queue and replacement in the sort are $O(1)$ operations.

On a typical, modest query (“boy” “play” “park”) on the default ConceptNet database, we allow acti-

Small ConceptNet		**P4-3.4 GHz**			**XC2V6000**					
					8 FPGAs (128 PEs)			40 FPGAs (512 PEs)		
Query	initial nodes	edges visited	% active	query time	edges visited	query time	per FPGA speedup	edges visited	query time	per FPGA speedup
"run marathon"	1	370	0.45	75 μs	81K	15 μs	0.63	81K	4.3 μs	0.43
"boy" "play" "park"	3	4600	5.6	0.99 ms	81K	15 μs	8.3	81K	4.3 μs	5.6
"person" "play" "dog" "park"	4	22K	27	3.4 ms	81K	15 μs	28	81K	4.3 μs	20
NYT-Abramoff article	109	23K	28	3.5 ms	81K	15 μs	29	81K	4.3 μs	20

Table 2.2: Comparison of Query Execution Times on Small ConceptNet Database

Default ConceptNet		**P4-3.4 GHz**			**XC2V6000** 176 FPGAs (2048 PEs)			
Query	initial nodes	edges visited	% active	query time	edges visited	query time	total speedup	per FPGA speedup
"run marathon"	1	450K	27	94 ms	1.6M	19 μs	4900	28
"boy" "play" "park"	3	540K	32	110 ms	1.6M	19 μs	5800	33
"person" "play" "dog" "park"	4	920K	55	190 ms	1.6M	19 μs	10000	57
NYT-Abramoff article	109	930K	56	190 ms	1.6M	19 μs	10000	57

Table 2.3: Comparison of Query Execution Times on Default ConceptNet Database

vation to spread for three steps and visit 539,819 edges. Each edge visit takes about 700 cycles (around 200 ns) including one cache miss to main memory, which accounts for roughly 300 of the 700 cycles. On average, this includes 12 L1 cache misses that are serviced by the L2 cache at 20 cycles apiece. All told, the query takes over 386,841,905 cycles, or about 113 ms. This query starts with three graph nodes activated, so the first few graph steps have moderate activity as activation spreads out from the initial nodes. Queries that start with many initial terms or high fanout nodes, as is typical in document processing tasks, will start with more of the graph active and consequently visit more nodes and require greater runtime (*e.g.*, the NYT query in Table 2.2).

To collect data for the sequential implementation, we compiled the code with GCC 3.4.1 with the -O3 option and ran it on a 3.4 GHz Pentium-4 Xeon machine. We used the Pentium cycle counters to capture complete runtime. Separate non-timing runs were used to collect basic statistics on edges visited. Cache statistics were captured with the Pentium event counters using PAPI-3.2.1 [BDG$^+$00, PAP06].

Tables 2.2 and 2.3 summarize the results from several typical ConceptNet queries.

2.7.4 FPGA Implementation

For the FPGA implementation, we place graph nodes into Block RAMs and built a specialized processing engine for ConceptNet spreading activation, which is pipelined to handle one edge operation per cycle. Each such processing engine requires 320 Virtex-2 slices. We exploit the dual-port capabilities of the Block RAM to perform a read of the current graph node state, compute an activity update, and write back the graph node state in the edge-update pipeline. We connect graph-processing engines together with a packet-switched or time-multiplexed overlay network (*i.e.*, network-on-a-Chip—see companion paper [KMd$^+$06]). The processing engine and network operate at 166 MHz (XC2V6000-4). To avoid serial bottlenecks on node processing, we decompose large nodes, those with high fanin or fanout, into a set of edge-limited nodes using fanin and fanout trees to preserve the original graph connectivity. To minimize network contention, we place graph nodes onto processing engine-memory block pairs to maximize locality using an efficient partitioner (UMpack's multilevel partitioner, `UCLA_MLPart4.21.1` [CKM00]) similar to [dD05].

In the simplest case, we use a time-multiplexed network and process every graph node and every edge on every graph step. That is, we do not exploit activity sparseness. Note that since each edge update occurs in pipelined fashion, we spend two cycles processing each edge (one sending and one receiving) for a total of 12 ns (XC2V6000-4) compared to the 200 ns per edge for the processor. Further, we get multiple processing engines per FPGA (*e.g.*, 32 on an XC2V6000), so that we get two to three orders of magnitude higher edge-processing throughput on the FPGA than on the processor. Since the FPGA implementation processes every edge, it processes an order of magnitude more edges than the processor in modest queries like ("boy" "play" "park"); however, it takes no more time to process compound queries that start with more initial terms (see Tables 2.2 and 2.3).

Each ConceptNet edge can be represented in 32b. Assuming that we can use only a power of two number of Block RAMs, we use 128 of the 144 Block RAMS on the XC2V6000. This gives us $128 \times 512 = 64$K edges per XC2V6000. Consequently, it will take at least 16 leaf FPGAs to hold the default ConceptNet.

Our FPGA performance numbers are calculated from a mapped implementation for the key elements (processing engine and network switches) and a cycle-accurate schedule of a graph step. We mapped our processing engine and network switches to an XC2V6000-4 and validated 166 MHz operation. On one XC2V6000, we get 32 processing engines using a Butterfly fat tree (BFT) interconnect structure (see Table 2.4). At the root of the leaf FPGAs, we have 4 input and 4 output channels. We use dedicated route FPGAs with 4 input and output downlinks and two input and output uplinks to continue to connect the leaf FPGAs up into a $p \approx 0.5$ BFT (see Figure 2.3 and Table 2.5). Based on timing from this implementation (*e.g.*, cycles per switch, pipeline stages in the processing engine), we completely scheduled computation and communication in a single graph step for a given number of processors and network organization [KMd$^+$06].

2.7.5 Discussion

As shown in Tables 2.2 and 2.3, the reconfigurable implementation gets at least an order of magnitude speedup per FPGA compared to the processor solution for modest queries. Further, the FPGA shows ex-

Component	**#**	**Slices Each**	**Total Slices**	**% Area**
Processing Engines	32	320	10240	30%
Node Address Memory	448 max graph nodes/PE	12/node	5376	16%
BFT Switches			3630	11%
L1	16 π	96	1536	
L2	16 T	72	1152	
L3	8 π	96	768	
L4	8 T	72	576	
L5	4 T	72	288	
TM Memory	1600 max cycles supported	9/cycle	14400	43%
Total			33646	100%

Table 2.4: Breakdown of Logic in ConceptNet Leaf FPGA with 32 PEs (XC2V6000)

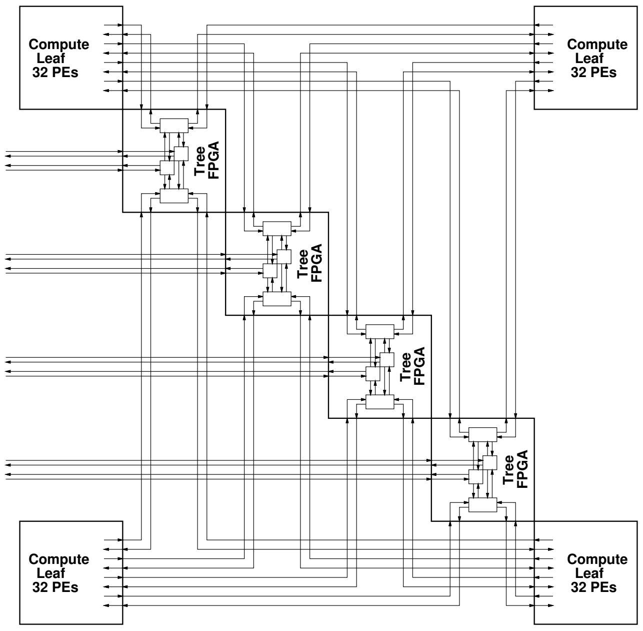


Figure 2.3: BFT Network with 128 PEs in 8 FPGAs

	FPGAs		
Total PEs	Compute Leaves	Tree Interconnect	Total
128	4	4	8
512	16	$4\times4+8=24$	40
2048	64	$4\times24+16=112$	176

Table 2.5: Multichip BFT Composition

cellent scaling to tens and hundreds of FPGAs, whereas the processor version will not scale as nicely. For compound queries, the advantage per FPGA increases. For the simple queries with low activity, it may be possible to also exploit sparse activity using packet-switched interconnect to further reduce the FPGA runtime (see [KMd$^+$06]).

2.8 Variations and Future Work

The applications outlined so far have all worked on static graphs. That is, we know the graph before the computation starts and the graph does not change during the computation. Further, since the graphs are known, we can place the tasks offline for spatial locality. Note that placement and routing are graph algorithms, so we expect to be able to use the same machine for placement and routing of the graph as we use to run the graph algorithms.

One generalization for future work is to efficiently support graph algorithms where the graph changes during the computation, that is, allow nodes and edges to be added and removed from the graph. In addition to allowing support for the new nodes, this will demand online placement of the new nodes and routing of the new links.

Many applications have mostly static graphs. That is, the graph may be large (millions of nodes and edges), but only a few edges are changed at a time. One example is a large knowledge base that filters out facts and adds new facts (nodes and edges) as it identifies facts that are not already in the knowledge base. Another example is a learning-based SAT solver (*e.g.* [MSS99, ZMMM01]). In these SAT solvers, the learned clause database becomes large (hundreds of thousands to millions of entries); however, there will be many graph operations per conflict and each conflict adds only a few clauses to the database. Consequently, we are changing only a tiny fraction (maybe 0.001%) of the graph at a time.

As noted in Section 2.7 our primary comparison is to a static, time-multiplexed GraphStep implementation. For low activities, a dynamic version might be more efficient. Further, low activities and evolving graphs might motivate adaptive techniques for graph node placement, perhaps moving nodes based on dynamic activity to enhance locality and parallelism.

2.9 Related Work

The idea of integrating computing with memory certainly is not new [LRSS84, DFK$^+$92, PAC$^+$97, Mar97, OCS98, PJA$^+$99, SMK$^+$96]. What is new is a suitable concurrent system architecture that organizes applications to exploit the parallelism and high memory bandwidth of these hardware architectures. As already noted in Section 2.6.3 many existing or proposed multiprocessor and PIM architectures could be useful implementation targets.

Some efforts to explore logic and DRAM integration have been focused around other concurrent system architectures. Active Pages [OCS98] was designed to support a data parallel model that specifically did not efficiently handle interconnect between pages. Vector IRAM [KP02] supported a vector model, making it suitable for dense applications, but not necessarily efficient for irregular, sparse-graph applications.

The GraphStep system architecture follows the vision of Hillis' Connection Machine (CM) [Hil85]. The CM was an early herald of the data parallel system architecture [HS86], and the first Connection Machines were SIMD implementations. As Figure 2.1 suggests, GraphStep is a refinement and restriction on the data parallel system architecture to more directly and efficiently support parallel graph algorithms.

2.10 Conclusions

The high bandwidth and low latency available from the small, distributed, on-chip memories in modern FPGAs provide another opportunity for delivering high performance with field-programmable custom computing machines. This opens up the opportunity for these machines to accelerate a distinct and complementary class of applications to those that traditionally exploit the high computational throughput of FPGAs and reconfigurable architectures. We can capture many of these data-intensive applications with a sparse, graph-oriented concurrent system architecture. We show how we can use this system architecture to exploit the high memory performance of these machines to deliver performance that is orders of magnitude better than that of microprocessors on these memory-bound applications.

Chapter 3

The Dishoom Reconfigurable Compute Platform

The Dishoom platform, which provides the hardware implementation for MATTER, is organized into a series of tiles, as shown in Figure 3.1. Figure 3.2 shows a close-up of a single board. The major components of each tile are a Xilinx XC2V6000-4 FPGA, Xilinx XC2C512 Complex Programmable Logic Device (CPLD), Arcturus Networks uC-DIMM Coldfire 5272, 128 Mb of Intel FLASH memory, and 512 Mb of Micron DDR-SDRAM. The uC-DIMM, FLASH, and CPLD are primarily used to program the FPGA. The FPGA will be used to implement MATTER, while the DDR-SDRAM serves as a high-capacity off-chip store to complement the on-chip memory. It can also hold data in stasis while the FPGA is reconfigured dynamically.

The Dishoom platform increases vastly in usefulness when tiles are networked together. The tops and bottoms of the tiles each have four HSEC8 high-speed connectors from Samtec, one on each edge. When connected on all four sides, the tiles form a network similar to a 3D Manhattan Mesh, albeit one where each layer is offset from the one below it. Each connector provides 36 unidirectional signals between FPGAs, at a speed of 200 MHz, or 7.2 Gb/s.

The Dishoom Virtex 2 FPGA can be programmed over ethernet through the uC-DIMM, which is connected via the CPLD to the FLASH memory. This memory can hold as many as five configuration files for the Virtex 2 FPGA. The CPLD acts as a programming interface between the FLASH and the FPGA. The FPGA-CPLD connection also lets the FPGA communicate with the uC-DIMM, and can be used to trigger mid-execution reconfiguration. Configuration files are delivered through the uC-DIMM through ethernet, over the Internet or a local intranet. The secondary programming interface is a JTAG chain, available mainly as a backup and debugging interface.

Figure 3.1: Tiled MATTER Dishoom Board

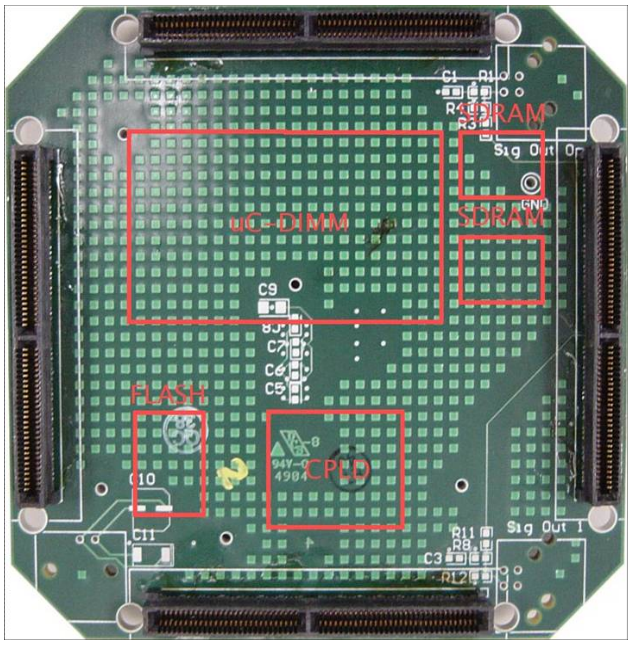


Figure 3.2: MATTER Dishoom Board

Chapter 4

Nanowire Chemistry: Dielectrophoretic assembly, reconfiguration, and disassembly of nanowire interconnects

This section describes the nanowire chemistry work peformed by Prof. Charles M. Lieber and Alexander D. Wissner-Gross, at Harvard University. (The connections between this work and the MATTER architecture are discussed in Section 1.2.)

We report the dielectrophoretic assembly, reconfiguration, and disassembly of heavily doped silicon nanowire interconnects in benzyl alcohol. Electrode pairs with high field enhancement factors enable the assembly of up to 50-$\mu\mathrm{m}$-long single-nanowire interconnects, and electrical transport measurements indicate that the nanowires function as 1-MΩ resistances. Phase modulation of one electrode in a set causes nanowires to reversibly reconfigure between the electrode tips. Moreover, multi-electrode phase modulation allows parallel reconfiguration of proximal interconnects. Field simulations indicate that this reconfiguration method can potentially scale to approximately 30 kHz switching speeds. For disassembly of interconnects, short high-voltage pulses trigger thermal detonation. The controllable reconfiguration of electronic nanostructures in situ opens up opportunities for colloidal, nanoelectromechanical connection architectures with synapse-like plasticity.

Programmability in electronic systems originates from the ability to form and reform nonvolatile connections. Devices in modern programmable architectures, such as FPGAs, typically derive this ability from controlled internal changes in material composition (antifuses) or charge placement (EPROM and flash) [Rose93]. However, for bottom-up nanoelectronics applications it may be advantageous to derive programmability from the manipulation of mobile components in addition to their internal states. Novel potential applications for which traditional device immobility is disadvantageous include dense arrays of nanostructure- mediated artificial synapses, breadboards for rapid prototyping of nanostructure circuits, and fault-tolerant logic in which individual components can be replaced automatically from a reservoir. In this report we take the first step of demonstrating that the simplest nanoelectronic components and interconnects can be assembled, reconfigured, and disassembled by an electromechanical process.

Various techniques for manipulating electronic nanostructures have been developed, including optical [Ritesh05], mechanical [Yu01], and electrical [Jang05,Dong05,Lieber01,Chen05,Bashir03,Harnack03] methods. Electromagnetic field switching is especially attractive for inexpensive parallel manipulation, and low-frequency near-field manipulation of neutral structures by dielectrophoresis is potentially less expensive than optical manipulation because of the low cost of semiconductor processing. Dry dielectrophoresis has been used to make a carbon nanotube switch [Jang05], but components cannot be replaced by this method. Wet dielectrophoresis has been previously used to trap a variety of structures, including NiSi nanowires [Dong05], CdS nanowires [Lieber01], carbon nanotubes [Chen05], silicon blocks [Bashir03], and ZnO nanorods [Harnack03]. Post-deposition electrical transport measurements were performed after drying the substrate [Bashir03,Harnack03], making reconfiguration impossible because of van der Waals pinning, or performed over an uncontrolled large number of pinned parallel interconnects [Chen05]. In this work, for the first time we demonstrate that dielectrophoresis may be used to reconfigure and disassemble nanoelectronic devices and that this process is compatible with simultaneous electrical transport.

Near-degenerately p-doped silicon nanowires were grown by established methods [Growth,Yi01] and then filtered and suspended in benzyl alcohol to remove highly polarizable, free gold catalyst particles. The nanowire growth wafer was sonicated lightly in isopropanol for 1 min. The suspension was vacuum filtered using a 12-μm mesh (Millipore Isopore). The filter mesh was sonicated in isopropanol, and the suspension was again filtered. The second filter mesh was sonicated in benzyl alcohol for 2 min and the suspension was used for trapping experiments. (Doped Si nanowires were grown using 20- to 150-nm diameter Au nanocluster catalysts, and $\mathrm{SiH_4}$ reactant (99.7%) and $\mathrm{B_2H_6}$ dopant (0.3%) in He (100 ppm), at 450 torr and $450^{\circ}\mathrm{C}$. Growth was performed for 10 to 60 min to achieve desired nanowire lengths.)

As a solvent for reconfiguration, benzyl alcohol has the advantages of being relatively viscous ($\mu \approx 5.47\mathrm{cP}$) and thus inhibiting nanowire motion in the absence of a field [CRC02]. It is nontoxic, protic (allowing stable suspension of silicon nanowires over days), and has a low vapor pressure. For long-term prevention of nanowire aggregation, it is especially attractive because its permittivity, $\epsilon_s \approx 11.9\epsilon_0$, is almost index-matched to the permittivity of bulk silicon, $\epsilon_{nw} \approx 12.1\epsilon_0$ [CRC02]. Silicon was selected to demonstrate potential compatibility of our technique with the assembly of more complex devices, such as axial

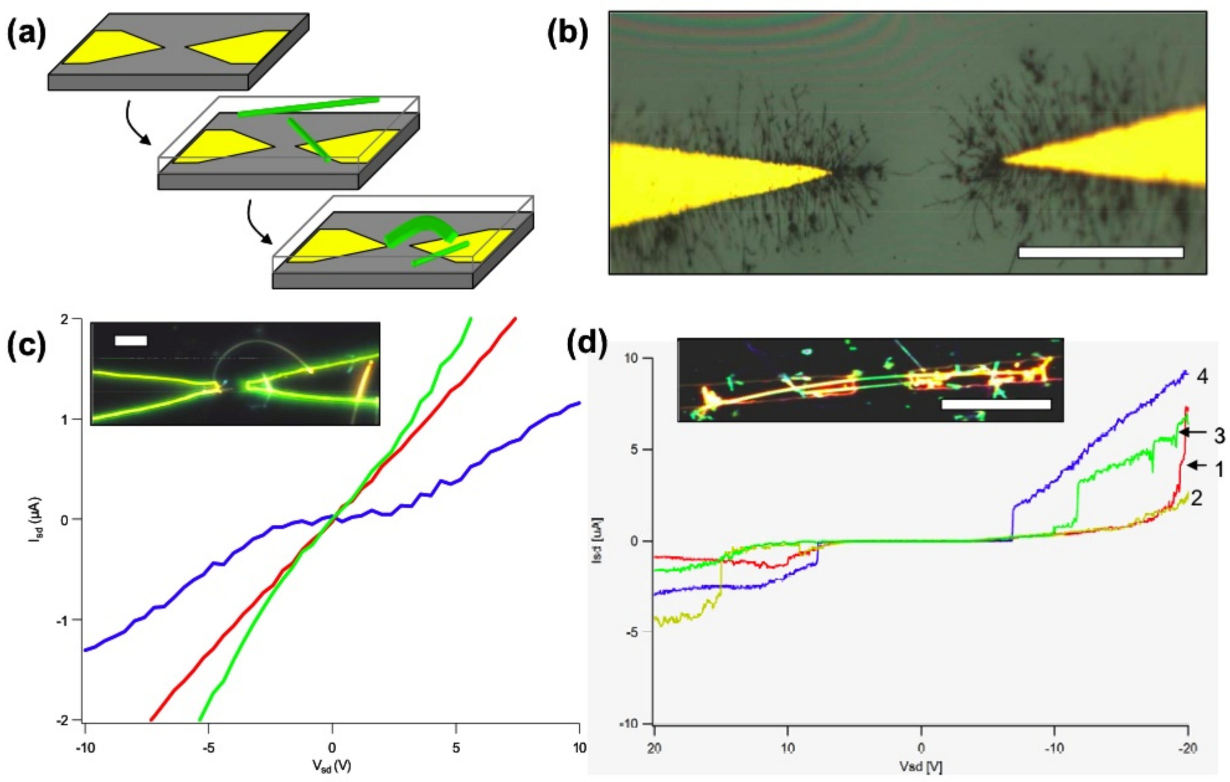


Figure 4.1: Dielectrophoretically assembled single-nanowire interconnects. (a) Schematic illustration of single-nanowire trapping process. (b) Light microscope image of individual nanowire stably trapped between electrodes separated by 50 μm. Scale bar is 50 μm. (c) AC electrical transport curves before (red) and after (green) nanowire trapping. Calculated parallel transport through nanowire is also shown (blue). Inset, single nanowire trapped by a 10-μm gap. Scale bar is 10 μm. (d) Dry transport curves of nanowires trapped from ethanol. Successive sweep numbers are indicated. Scale bar is 20 μm.

heterostructures [Gudiksen02]. In contrast, carbon nanotubes generally must be chemically functionalized to prevent aggregation, which can diminish their electrical conductivity [Zhang05].

Trapping experiments were performed with 100- to 250-nm-thick Au/Cr electrodes on a silicon wafer with a 200-nm oxide, to prevent shorts. Thicker electrodes were found to better allow nanowires to migrate along their edges toward the trapping region, most likely because of their reduced fringing fields, while thinner electrodes caused nanowires to make larger contact with the top faces of the electrodes. The electrode material was selected primarily to avoid oxidative damage and not by Schottky barrier considerations, since adsorption to electrodes would leave a large contact resistance regardless. Each electrode tapered to a tip at a 10^{o} angle with a 0.5- to 2.5-μm radius of curvature and a field enhancement factor of ~ 150, in order to preferentially trap nanowires at the tip.

The nanowire suspension was pipetted onto the electrode chip to form a 100- to 500-μm-thick film [Figure 4.1(a)]. For single-nanowire trapping and generally, electrode pairs were biased at 10 kHz, which lies above the solvent electrolysis frequency but minimizes parasitic capacitance effects. The bias was modulated

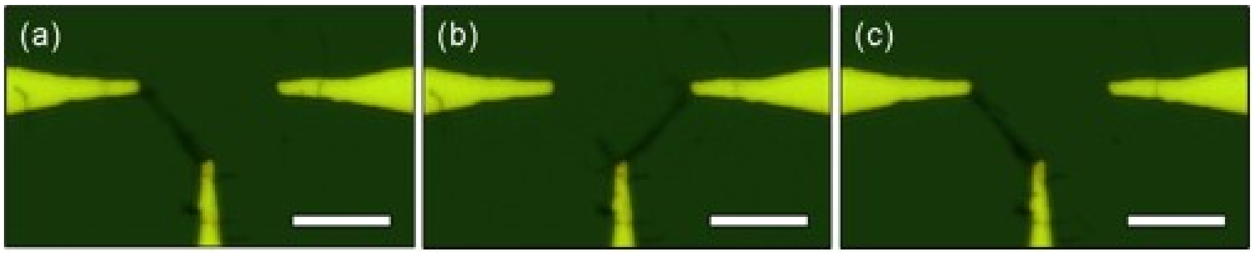


Figure 4.2: Reconfiguration of a single nanowire bundle. (a-c) Light microscope images of three-electrode planar reconfiguration, as the phase of the middle electrode is modulated. Scale bars are 15 μm. (A movie is available in the supplementary information CD-ROM that accompanies this report).

into 10-ms bursts with a period of 100 to 250 ms, which allowed hysteretic migration of nanowires toward the trapping region while minimizing "burn-in" from nanowires permanently conforming to an electrode. The amplitudes of the bursts were varied linearly as a function of desired nanowire length in order to keep power dissipation per unit length along the nanowire constant, and individual nanowires with lengths up to 50 μm were thus stably trapped [Figure 4.1(b)]. After averaging over transport hysteresis and subtracting the parallel solvent conductance, it was found that the nanowires behave as switchable $1\text{M}\Omega/\mu$m resistances with a $\sim$ 3 V built-in potential consistent with the work-function explanation [Figure 4.1(c)]. Transport measurements occurred at lower voltages than trapping, so nanowire movement is minimal. Trapping nanowires from ethanol and then allowing the substrate to dry showed that the wires indeed act as 200-400 kΩ/μm resistances [Figure 4.1(d)] with a sharp current turn-on at biases of 7-15 V, suggestive of electromechanical switching behavior.

Nanowire interconnect reconfiguration was achieved by modulating the phase of a third electrode [Figure 4.2], locking it opposite the phase of the electrode to which the interconnect is desired. After each reconfiguration, electrical transport between the third and first electrode was measured with a 20 V peak-to-peak sawtooth wave bias, in order to exceed the metal work function, and at 10 Hz, in order to slow electrolysis.

In addition to the reconfiguration of nanowires between adjacent gaps, it is possible both to manipulate larger numbers of interconnects in parallel and to completely remove an interconnect. Parallel reconfiguration of nanowire interconnects between shared electrodes was achieved by modulating the locked phases of multiple electrodes [Figure 4.3].

Given the average field intensity gradient in this system, it is possible to estimate how rapidly a nanowire might be dielectrophoretically reconfigured. Consider a Stokes flow model for reconfiguration of a nanowire. The drag coefficient for an infinitely long cylinder [Tritton88] is given by

$$C_D \equiv \frac{f_D}{\frac{1}{2}\rho_{nw}u^2 d} \approx \frac{8\pi}{\mathrm{Re}(2.002 - \ln\mathrm{Re})} \ ,$$

where f_D is the drag force per unit length, ρ_{nw} is the cylinder density, u is velocity, d is the cylinder

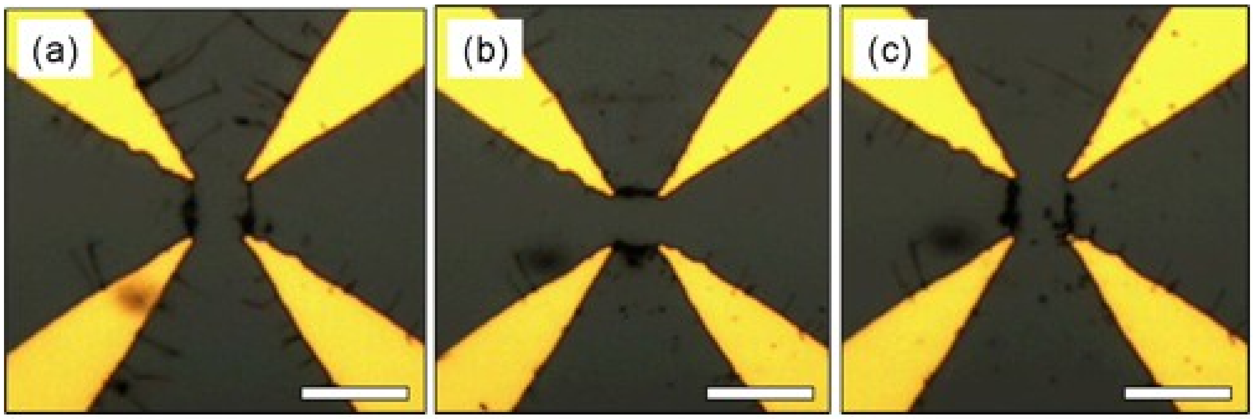


Figure 4.3: Parallel reconfiguration of nanowires. (a-c) Light microscope images of parallel reconfiguration of nanowires among four electrodes. The phases on the upper-right and lower- left electrodes are equal, and are modulated to induce the reconfiguration. Scale bars are 20 μm. (A movie is available in the supplementary CD-ROM).

diameter, $\mathrm{Re} \sim d\rho_s u/\mu$ is the Reynolds number, ρ_s is the solvent density, and μ is the dynamic viscosity. Two types of dielectrophoretically induced motion are observed: motion parallel and perpendicular to the nanowire axis. Under the mean-field approximation, the dielectrophoretic force per unit length along the field intensity gradient is

$$f_{DEP} = \frac{\epsilon_s \pi d^2}{8} \mathrm{Re}\left\{ \vec{K}(f) \cdot \vec{\bigtriangledown}(\vec{E}^2) \right\} \quad ,$$

where ϵ_s is the solvent permittivity and $\vec{K}(f)$ is the Clausius-Mossotti factor. For a cylindrical nanowire, the Clausius-Mossotti components are approximately

$$K_{\|}(f) \cong \frac{\epsilon_{nw}^* - \epsilon_s^*}{\epsilon_s^* + (\epsilon_{nw}^* - \epsilon_s^*)(1 - (1 + (d/2)^2/L^2)^{-1/2})} \approx \frac{\epsilon_{nw}^* - \epsilon_s^*}{\epsilon_s^*} \; (d << L)$$

and

$$K_{\perp}(f) \cong \frac{\epsilon_{nw}^* - \epsilon_s^*}{\epsilon_s^* \left(1 - \frac{\pi}{8}\right) + \epsilon_{nw}^* \left(\frac{\pi}{8}\right)}$$

parallel and perpendicular to the nanowire axis, respectively, where $\epsilon^* \equiv \epsilon_{nw,s} - i\frac{\sigma_{nw,s}}{2\pi f \epsilon_0}$ are the complex permitivities of nanowire and solvent, and L is the nanowire length. For the materials in this experiment, the remaining relevant electrohydrodynamic values are the densities $\rho_s \approx 1.04\mathrm{g\ cm}^{-3}$ and $\rho_{nw} \approx 2.33\mathrm{g\ cm}^{-3}$ [CRC02]. Nanowires were doped near the metallic limit [Lieber00] so it is estimated that $\sigma_{nw} \sim 1S/m$ and, in this non-electrolytic context, the solvent is assumed to be nonconductive ($\sigma_s \sim 0$). The terminal velocity u during switching is found numerically, by matching forces, to be $\sim 0.6\mathrm{m/s}$, suggesting a 30-kHz reconfiguration frequency for 10-μm displacements.

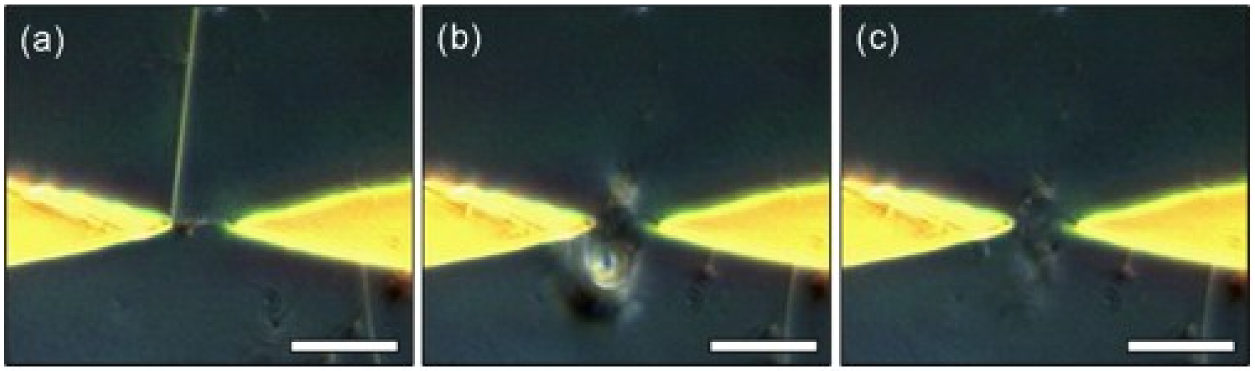


Figure 4.4: Disassembly of nanowire interconnects by high-voltage detonation. (a) Stably trapped nanowire before detonating voltage pulse. (b) Vapor bubble resulting from thermal detonation. (c) Only submicron fragments remain, and region is cleared to trap a new nanowire. Scale bars are 20 μm.

Interconnect “disassembly”was accomplished with 10-ms bursts at 110 V peak-to-peak for 10-μm electrode spacing [Figure 4.4]. The estimated current density under these conditions is as high as $\sim 5 \times 10^{10}\mathrm{A} \cdot \mathrm{m}^{-2}$, which is consistent with thermal detonation. Together with the ability to assemble and reconfigure colloidal electronic nanostructures, the disassembly of electronic nanostructures is reminiscent of receptor trafficking for synaptic plasticity [Manilow02].

Chapter 5

Activities and Additional Material

5.1 Activities

The group has performed the following activities under this contract:

- Kickoff meeting in Boston, January 7, 2005
- Bi-weekly videoconferences between Caltech, MIT, and SRI
- Visits to Charles Lieber's lab at Harvard
- André deHon (Caltech) visit to SRI
- Ian Eslick (MIT) visit to Caltech
- DARPA review at Boston (Harvard), March 16, 2005, including tour of nanowire lab
- MATTER retreat in Santa Barbara, California
- Tomás Uribe (SRI) visit to Caltech
- ACIP PI meetings in Monterey, California, and Marco Island, Florida

5.2 Software development

- MIT developed a LISP reference implementation of the ConceptNet algorithm
- MIT developed LISP infrastructure to simulate the concurrent hardware operations
- Caltech and SRI developed two C reference implementations

The SRI team was given access to CVS and SVN source code control repositories set up at Caltech and MIT.

Bibliography

[AI87] Arvind and R. A. Ianucci. Two fundamental issues in multiprocessing. In *Proceedings of DFVLR Conference on Parallel Processing in Science and Engineering*, pages 61–88, West Germany, June 1987.

[AJ89] Andrew Appel and T. Jim. Continuation-passing, closure-passing style. In *Proceedings of the ACM Conference on Principles of Programming Languages*, pages 293–302, 1989.

[Alt05] Altera Corporation, 2610 Orchard Parkway, San Jose, CA 95134-2020. *Stratix II Device Handbook*, 4.0 edition, December 2005.

[Bashir03] Lee, S. W.; Bashir, R. *Appl. Phys. Lett.* 2003, 83, 3833.

[BDG$^+$00] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci. A portable programming interface for performance evaluation on modern processors. *The International Journal of High Performance Computing Applications*, 14(3):189–204, 2000.

[BFA96] Jonathan Babb, Matt Frank, and Anant Agarwal. Solving graph problems with dynamic computational structures. In *Proceedings of SPIE: High-Speed Computing, Digital Signal Processing, and Filtering Using Reconfigurable Logic*, volume 2914, pages 225–236, November 1996.

[CCH$^+$00] Eylon Caspi, Michael Chu, Randy Huang, Nicholas Weaver, Joseph Yeh, John Wawrzynek, and André DeHon. Stream Computations Organized for Reconfigurable Execution (SCORE): Introduction and Tutorial. <`http://www.cs.berkeley.edu/projects/brass/documents/score_tutorial.html`>, short version appears in FPL'2000 (LNCS 1896), 2000.

[Chen05] Chen, Z.; Yang, Y.; Chen, F.; Qing, Q.; Wu, Z.; Liu, Z. *J. Phys. Chem.* B. 2005, 109, 11420.

[CKM00] Andrew Caldwell, Andrew Kahng, and Igor Markov. Improved Algorithms for Hypergraph Bipartitioning. In *Proceedings of the Asia and South Pacific Design Automation Conference*, pages 661–666, January 2000.

[CLR90] Thomas Cormen, Charles Leiserson, and Ronald Rivest. *Introduction to Algorithms*. MIT Press, 1990.

[CRC02] Lide, D. R., ed. *CRC Handbook of Chemistry and Physics*; 82nd ed.; CRC Press: New York, 2001, pp. 3-52, 6-163, 12-59.

[DAd+04] André DeHon, Joshua Adams, Michael deLorimier, Nachiket Kapre, Yuki Matsuda, Helia Naeimi, Michael Vanier, and Michael Wrighton. Design Patterns for Reconfigurable Computing. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 13–23, April 2004.

[dD05] Michael deLorimier and André DeHon. Floating-Point Sparse Matrix-Vector Multiply for FPGAs. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays*, pages 75–85, February 2005.

[DeH96] André DeHon. Reconfigurable Architectures for General-Purpose Computing. AI Technical Report 1586, MIT Artificial Intelligence Laboratory, 545 Technology Sq., Cambridge, MA 02139, October 1996.

[DeH00] André DeHon. The Density Advantage of Configurable Computing. *IEEE Computer*, 33(4):41–49, April 2000.

[DFK+92] William J. Dally, Stuart J. A. Fiske, John S. Keen, Richard A. Lethin, Michael D. Noakes, Peter R. Nuth, Roy E. Davison, and Gregory A. Fyler. The message-driven processor: A multicomputer processing node with efficient mechanisms. *IEEE Micro*, pages 23–39, April 1992.

[DHW02] André DeHon, Randy Huang, and John Wawrzynek. Hardware-Assisted Fast Routing. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 205–215, April 2002.

[dKM+06] Michael deLorimier, Nachiket Kapre, Nikil Mehta, Dominic Rizzo, Ian Eslick, Raphael Rubin, Tomás E. Uribe, Thomas F. Knight, Jr., and André DeHon. Graphstep: A system architecture for sparse-graph algorithms. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*. IEEE, 2006. To appear.

[Dong05] Dong, L.; Bush, J.; Chirayos, V.; Solanki, R.; Jiao, J.; Ono, Y.; Conley, J. F., Jr.; Ulrich, B. D. *Nano Lett.* 2005, 5, 2112.

[Fah79] Scott E. Fahlman. *NETL: A System for Representing and Using Real-World Knowledge*. MIT Press, 1979.

[FM82] C. M. Fiduccia and R. M. Mattheyses. A linear time heuristic for improving network partitions. In *Proceedings of the 19th Design Automation Conference*, pages 175–181, 1982.

[Gudiksen02] Gudiksen, M. S.; Lauhon, L. J.; Wang, J.; Smith, D.; Lieber, C. M. *Nature* 2002, 415, 617.

[Harnack03] Harnack, O.; Pacholski, C.; Weller, H.; Yasuda, A.; Wessels, J. M. *Nano Lett.* 2003, 3, 1097.

[HBS73] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular actor formalism for artificial intelligence. In *Proceedings of the International Joint Conference on AI*, 1973.

[Hil85] W. Daniel Hillis. *The Connection Machine*. MIT Press, 1985.

[HJ92] Dana S. Henry and Christopher F. Joerg. A tightly-coupled processor-network interface. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 1992.

[HS86] W. Daniel Hillis and Guy L. Steele. Data parallel algorithms. *Communications of the ACM*, 29(12):1170–1183, December 1986.

[HS95] Andrew S. Huang and John P. Shen. A limit study of local memory requirements using value reuse profiles. In *Proceedings of MICRO-28*, pages 71–91, December 1995.

[Hua04] Randy Ren-Fu Huang. *Hardware-Assisted Fast Routing for Runtime Reconfigurable Computing*. PhD thesis, University of California at Berkeley, 2004.

[HWD03] Randy Huang, John Wawrzynek, and André DeHon. Stochastic, Spatial Routing for Hypergraphs, Trees, and Meshes. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays*, pages 78–87, February 2003.

[Int05] Intel Corporation. Intel Pentium 4 processor product briefs. <`http://www.intel.com/design/Pentium4/prodbref/`>, December 2005.

[Jang05] Jang, J. E.; Cha, S. N.; Choi, Y.; Amaratunga, G. A. J.; Kang, D. J.; Hasko, D. G.; Jung, J. E.; Kim, J. M. *Appl. Phys. Lett.* 2005, 87, 163114.

[KM93] Jun-Tae Kim and Dan I. Moldovan. Classification and retrieval of knowledge on a parallel marker-passing architecture. *IEEE Transactions on Knowledge and Data Engineering*, 5(5):753–761, October 1993.

[KMd$^+$06] Nachiket Kapre, Nikil Mehta, Michael deLorimier, Raphael Rubin, Henry Barnor, Michael J. Wilson, Michael Wrighton, and André DeHon. Packet-switched vs. time-multiplexed fpga overlay networks. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*. IEEE, 2006.

[KMH01] Tomoyoshi Kobori, Tsutomu Maruyama, and Tsutomu Hoshino. A cellular automata system with fpga. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, 2001.

[KP02] Christoforos Kozyrakis and David Patterson. Vector vs superscalar and vliw architectures for embedded multimedia benchmarks. In *Proceedings of the International Symposium on Microarchitecture*, pages 283–293, 2002.

[LDK$^+$98] Whay Sing Lee, William J. Dally, Stephen W. Keckler, Nicholas P. Carter, and Andrew Chang. An efficient, protected message interface. *IEEE Computer*, 31(11):69–75, November 1998.

[Lei92] Frank Thomson Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*. Morgan Kaufmann Publishers, Inc., 1992.

[Lieber00] Cui, Y.; Duan, X.; Hu, J.; Lieber, C. M. *J. Phys. Chem. B.* 2000, 104, 5213.

[Lieber01] Duan, X.; Huang, Y.; Cui. Y.; Wang, J.; Lieber, C. M. *Nature*, 2001, 409, 66.

[LRSS84] Chris Lutz, Steve Rabin, Chuck Seitz, and Don Speck. Design of the mosaic element. In Paul Penfield, Jr., editor, *Proceedings, Conference on Advanced Research in VLSI*, pages 1–10, Cambridge, MA, January 1984.

[LS04] Hugo Liu and Push Singh. ConceptNet – A Practical Commonsense Reasoning Tool-Kit. *BT Technical Journal*, 22(4):211, October 2004.

[Mar97] Norm Margolus. An fpga architecture for dram-based systolic computations. In *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, pages 2–11, 1997.

[MAS$^+$97] Masato Motomura, Yoshiharu Aimoto, Atsufumi Shibayama, Yoshikazu Yabe, and Masakazu Yamashina. An embedded dram-fpga chip with instantaneous logic reconfiguration. In *Digest of Technical Papers Symposium on VLSI Circuits*, pages 55–56, 1997.

[McK04] Sally A. McKee. Reflections on the memory wall. In *Proceedings of Computing Frontiers*, April 2004.

[MCMB93] George Milne, Paul Cockshott, George McCaskill, and Peter Barrie. Realising massively concurrent systems on the space machine. In *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, pages 26–32, April 1993.

[ME95] Larry McMurchie and Carl Ebling. PathFinder: A Negotiation-Based Performance-Driven Router for FPGAs. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays*, pages 111–117. ACM, February 1995.

[MHH02] Oskar Mencer, Zhining Huang, and Lorenz Huelsbergen. Hagar: Efficient multicontext graph processors. In *Proceedings of the International Conference on Field-Programmable Logic and Applications*, pages 915–924, 2002.

[MP03] S. Macskassy and F. Provost. A simple relational classifier. In *Proc. of the KDD-2003 Workshop on Multirelational Data Mining*, 2003.

[Manilow02] Manilow, R.; Malenka, R. C. *Ann. Rev. Neuro*. 2002, 25, 103.

[MSS99] João P. Marques-Silva and Karem A. Sakallah. Grasp: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, May 1999.

[OCS98] Mark Oskin, Frederic T. Chong, and Timothy Sherwood. Active pages: a model of computation for intelligent memory. In *Proceedings of the International Symposium on Computer Architecture*, June 1998.

[PAB+06] Dac C. Pham, Tony Aipperspach, David Boerstler, Mark Bolliger, Rajat Chaudhry, Dennis Cox, Paul Harvey, Paul M. Harvey, H. Peter Hofstee, Charles Johns, Jim Kahle, Atsushi Kameyama, John Keaty, Yoshio Masubuchi, Mydung Pham, Jürgen Pille, Stephen Posluszny, Mack Riley, Daniel L. Stasiak, Masakazu Suzuoki, Osamu Takahashi, James Warnock, Stephen Weitzel, Dieter Wendel, and Kazuaki Yazawa. Overview of the architecture, circuit design, and physical implementation of a first-generation cell processor. *IEEE Journal of Solid State Circuits*, 41(1):179–196, January 2006.

[PAC+97] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K. Yelick. A case for intelligent ram: Iram. *IEEE Micro*, 17(2):34–44, Mar/Apr 1997.

[PAP06] PAPI Project. Performance application programming interface. <http://icl.cs.utk.edu/papi/>, January 2006.

[PJA+99] Stylianos Perissakis, Yangsung Joo, Jinhong Ahn, André DeHon, and John Wawrzynek. Embedded DRAM for a Reconfigurable Array. In *Proceedings of the 1999 Symposium on VLSI Circuits*, June 1999.

[Pohl78] Pohl, H. A. *Dielectrophoresis*; Cambridge University Press: Cambridge: 1978.

[Ritesh05] Agarwal, R.; Ladavac, K.; Roichman, Y.; Yu, G.; Lieber, C. M.; Grier, D. G. *Opt. Express* 2005, 13, 8906.

[Rose93] Rose, J., el Gamal, A., Sangiovanni-Vincentelli, A. *Proc. IEEE 1993*, 81, 1013.

[SCB+98] Allan Snavely, Larry Carter, Jay Boisseau, Amit Majumdar, Kang Su Gatlin, Nick Mitchell, John Feo, and Brian Koblenz. Multi-processor performance on the tera mta. In *Proceedings of Supercomputing*, November 1998.

[SG96] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

[SMK+96] Toshia Sunaga, Hisatada Miyatake, Koji Kitamura, Peter M. Kogge, and Eric Retter. A processor in memory chip for massively parallel embedded applications. *IEEE Journal of Solid State Circuits*, 31(10):1556–1559, October 1996.

[STO03] Ryan Schneider, Laurence Turner, and Michal Okoniewski. Application of fpga technology to accelerate the finite-difference time-domain (ftdt) method. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays*, pages 97–105, 2003.

[Tritton88] Tritton, D. J. *Physical Fluid Dynamics*; 2nd ed.; Oxford University Press: Oxford, 1988, p. 32 and references therein.

[Val90] Leslie G. Valliant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103, August 1990.

[WD03] Michael Wrighton and André DeHon. Hardware-Assisted Simulated Annealing with Application for Fast FPGA Placement. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays*, pages 33–42, February 2003.

[WF05] J.V. White and C.G. Fournelle. Threat detection for improved link discovery. In *International Conference on Intelligence Analysis*, 2005.

[WM95] Wm. A. Wulf and Sally A. McKee. Hitting the memory wall: Implications of the obvious. *Computer Architecture News*, 23(1):20–24, 1995.

[Wri03] Michael Wrighton. A Spatial Approach to FPGA Cell Placement by Simulated Annealing. Master's thesis, California Institute of Technology, June 2003.

[WTS+97] Elliot Waingold, Michael Taylor, Devabhaktuni Srikrishna, Vivek Sarkar, Walter Lee, Victor Lee, Jang Kim, Matthew Frank, Peter Finch, Rajeev Barua, Jonathan Babb, Saman Amarasinghe, and Anant Agarwal. Baring it all to software: Raw machines. *IEEE Micro*, 30(9):86–93, September 1997.

[Xil03] Xilinx, Inc., 2100 Logic Drive, San Jose, CA 95124. *Xilinx Virtex-II Platform FPGAs Data Sheet*, October 2003. DS031 <`http://direct.xilinx.com/bvdocs/publications/ds031.pdf`>.

[Xil05] Xilinx, Inc., 2100 Logic Drive, San Jose, CA 95124. *Xilinx Virtex-4 Family Ovreview*, June 2005. DS112 <`http://direct.xilinx.com/bvdocs/publications/ds112.pdf`>.

[Yi01] Cui, Y.; Lauhon, L. J.; Gudiksen, M. S.; Wang, J.; Lieber, C. M. *Appl. Phys. Lett.* 2001, 78, 2214.

[Yu01] Huang, Y.; Duan, X.; Wei, Q.; Lieber, C. M. *Science* 2001, 291, 630.

[Zhang05] Zhang, Z.-B.; Cardenas, J.; Campbell, E. E. B.; Zhang, S.-L. *Appl. Phys. Lett.* 1995, 87, 043110.

[ZMMM01] Lintao Zhang, Conor F. Madigan, Matthew H. Moskewicz, and Sharad Malik. Efficient conflict driven learning in a boolean satisfiability solver. In *Proceedings of the International Conference on Computer-Aided Design*, pages 279–285, 2001.

Appendix A

MATTER Graph Machine Design Space for Marker Passing

We sketch the design space for a Graph Machine targeted at Marker-Passing algorithms, and assess the performance potential and area costs.

A.1 Basic Marker Passing Algorithm

1. Broadcast initial facts/activation to all nodes
2. Repeat until no updates (reach fixed point)
   a. Marker-Pass-Step: For each graph node
      - Push marker(s) along all appropriate edges
3. Perform Reduce to collect all results

A.2 Key Operations

- Broadcast – send a message (invoke a method) on all graph nodes
- Marker-Pass-Step – perform one step = make local update and push results out all (appropriate) edges
- Reduce – collect results from all graph nodes

A.3 Parameters

Algorithm	I	Number of initial broadcast operations
	R	Number of results
Graph	V	Number of nodes in graph
	D	Diameter of graph
	p	Rent parameter characterizing locality of graph
Graph Implementation	B	Bits per graph node
Graph Machine	N	Number of processors
	M	Maximum number of graph nodes per processor
	P	Clock cycles per graph op on each graph node
	P_{bcst}	Clock cycles per broadcast op on each graph node
	P_{reduce}	Clock cycles per reduce op on each graph node
	C_{mem}	Memory capacity per node
Mapping Quality	γ	Memory filling factor
Technology	α	Clock cycles to cross one PE width (height)
	β	Clock cycles to cross chip boundary
	T_{clk}	Clock cycle
Timing	T_{alg}	Time for marker passing algorithm
	T_{bcst}	Time for initial broadcast
	T_{reduce}	Time for final reduce
	T_{mps}	Time to compute one marker-pass step
	T_{comp}	Time to perform compute operations
	T_{comm}	Time to perform communication
	T_{lat}	Latency for communication operation
	T_{load}	Load factor on communication network
		(number of cycles due to network bandwidth limitations)

A.4 Basic Relationships

Memory size and folding factor:

$$M = \gamma \left\lceil \frac{V}{N} \right\rceil \tag{A.1}$$

$$C_{mem} \geq B \times M \tag{A.2}$$

γ is our fudge factor for imperfect filling of nodes.

A complete marker passing algorithm involves the broadcasts, a set of marking-passing steps, and the reduce.

$$T_{alg} = T_{bcst} + D \times T_{step} + T_{reduce} \tag{A.3}$$

The longest distance to propagate is the diameter of the graph, D.

$$D \leq N \tag{A.4}$$

We called out P_{bcst}, P_{reduce} separately from P on the assumption that they are most likely smaller. Notably, a marker-step op may need to send something out to each of edges, while broadcast and reduce operations set only one thing or grab one result.

$$\begin{aligned} T_{bcst} &\approx I + T_{lat} + M \times P_{bcst} && \text{(A.5)} \\ T_{reduce} &\approx M \times P_{reduce} + T_{lat} + R && \text{(A.6)} \end{aligned}$$

$$\begin{aligned} T_{step} &\approx T_{comp} + T_{comm} && \text{(A.7)} \\ T_{comp} &\leq P \times M && \text{(A.8)} \\ T_{comm} &\geq \max(T_{lat}, T_{load}) && \text{(A.9)} \\ T_{lat} &\leq \alpha\sqrt{N} && \text{(A.10)} \end{aligned}$$

T_{lat} here is for a simple 2D configuration. Later, we will look at alternate and more sophisticated models.

We also assume that the node size, and hence α, is independent of M. This will not be a valid assumption across an extremely large M. Also, a better model (accounting for possible superlinear network growth) is to look at machine size in terms of area, and have a function for area in terms of N and other parameters (*e.g.* p).

A.5 Simple, Optimal Size Calculation

Our goal in this section is to keep it simple and demonstrate the basic calculations and optimizations.

Assuming that latency dominates:

$$T_{step} \approx P \times \gamma \left(\frac{V}{N}\right) + \alpha\sqrt{N} \tag{A.11}$$

Minimize by taking derivative and setting equal to zero:

$$P \times V \times \gamma \left(\frac{-1}{N^2}\right) + \left(\frac{\alpha}{2}\right) \frac{1}{\sqrt{N}} = 0 \tag{A.12}$$

$$P \times V \times \gamma \left(\frac{1}{N^2}\right) = \left(\frac{\alpha}{2}\right) \frac{1}{\sqrt{N}} \tag{A.13}$$

$$P \times V \times \gamma = \left(\frac{\alpha}{2}\right) N^{\left(\frac{3}{2}\right)} \tag{A.14}$$

$$\frac{2 \times P \times V \times \gamma}{\alpha} = N^{\left(\frac{3}{2}\right)} \tag{A.15}$$

$$N = \left(\frac{2 \times P \times V \times \gamma}{\alpha}\right)^{\left(\frac{2}{3}\right)} \tag{A.16}$$

E.g., consider $V = 10^5$, $\alpha = 1$, $P = 5$, $\gamma = 1$

$$N = \left(\frac{2 \times 5 \times 10^5 \times 1}{1}\right)^{\left(\frac{2}{3}\right)} = 10^4 \tag{A.17}$$

$$T_{step} \approx 5 \times 1 \left(\frac{10^5}{10^4}\right) + \sqrt{10^4} = 150 \tag{A.18}$$

$$M = 1 \times \left\lceil \frac{10^5}{10^4} \right\rceil = 10 \tag{A.19}$$

For a broadcast operation:

$$\begin{aligned} T_{bcst} &\approx I + T_{lat} + M \times P_{bcst} \\ &\approx I + \alpha\sqrt{N} + \gamma \left(\frac{V}{N}\right) P_{bcst} \end{aligned} \tag{A.20}$$

Taking the derivative and setting to zero will have similar structure with different constants:

$$N = \left(\frac{2 \times P_{bcst} \times V \times \gamma}{\alpha}\right)^{\left(\frac{2}{3}\right)} \tag{A.21}$$

Similarly for reduce.

For the full algorithm:

$$\begin{aligned} T_{alg} &= T_{bcst} + D \times T_{step} + T_{reduce} \\ &= I + \alpha\sqrt{N} + \gamma \left(\frac{V}{N}\right) P_{bcst} \\ &\quad + D \times \left(P \times \gamma \left(\frac{V}{N}\right) + \alpha\sqrt{N}\right) \\ &\quad + R + \alpha\sqrt{N} + \gamma \left(\frac{V}{N}\right) P_{reduce} \end{aligned} \tag{A.22}$$

$$= \quad I + R + \alpha\,(D+2)\,\sqrt{N} + \gamma\left(\frac{V}{N}\right)(D \times P + P_{bcst} + P_{reduce}) \tag{A.23}$$

We have the same powers of N, so only the constants change.

$$(D \times P + P_{bcst} + P_{reduce}) \times V \times \gamma\left(\frac{1}{N^2}\right) = \left(\frac{\alpha\,(D+2)}{2}\right)\frac{1}{\sqrt{N}} \tag{A.24}$$

$$\frac{(D \times P + P_{bcst} + P_{reduce}) \times V \times \gamma \times 2}{\alpha\,(D+2)} = N^{\left(\frac{3}{2}\right)} \tag{A.25}$$

$$N = \left(\frac{2 \times \gamma \times (D \times P + P_{bcst} + P_{reduce}) \times V}{\alpha\,(D+2)}\right)^{\left(\frac{2}{3}\right)} \tag{A.26}$$

A.6 Sequential Optimization and Optimized Sequential Performance Model

A simple marker-passing phase is a transitive closure computation. That is, we are looking for reachability between some starting points (*e.g.*, properties) and various nodes. In a transitive closure, we need to visit each node only once. Consequently, the basic algorithm and analysis above is inefficient in the extreme of a sequential implementation, and most likely wasteful even in the parallel case.

In a sequential implementation, we would make sure to visit each node at most once during a marker passing phase:

1. For each node:
   a. apply broadcast markers
   b. put nodes meeting activation criteria into work queue
2. While work queue not empty:
   a. pop a node
   b. propagate marker out all suitable edges
      - if node at end of edge not already marked, mark and add to work queue

A.6.1 Sequential Parameters

Graph Structure and Locality	E_a	Average number of active edges requiring a fetch
Technology	T_{fetch}	Time for a non-local fetch (miss in cache(s))
Architecture/Impl.	T_{set}	Time to set state based on broadcast
	T_{gop}	Time for graph operation
Timing	T_{serial}	Time for serial algorithm
	T_{ivisit}	Time for initial visits based on broadcast
	T_{mark}	Time for marker pass phase

A.6.2 Sequential Model

Here, our time is:

$$T_{serial} = N \times T_{ivisit} + S \times T_{mark} \tag{A.27}$$

We walk through the entire graph on the initial visit. If the database is large, it will not fit in the cache. We pay (at least) one expensive fetch bringing in the data item. If it is laid out well relative to the cache lines, perhaps prefetch brings in the rest of the graph node, so we pay only the one miss.

$$T_{ivisit} = T_{fetch} + I \times T_{set} + T_{gop} \tag{A.28}$$

We lump together the operations for graph handling (*e.g.*, procedure call overhead) into T_{gop}.

If the activated set is large, most of the graph fetches will be misses. As each of those follows links, they may result in cache misses; some will be recently visited, so they will not generate cache misses.

$$T_{mark} = T_{fetch} + E_a \times T_{fetch} + T_{gop} \tag{A.29}$$

E.g. $E_a = 1$, $T_{fetch} = 50$, $T_{gop} = 100$, $S = 0.5N$, $N = 10^5$:

$$S \times T_{mark} = 100 \times 10^5 = 10^7 \tag{A.30}$$

The marker passing phase in the previous example was

$$D \times T_{step} = D \times 150 \tag{A.31}$$

If $D = N$, our simple, parallel version is actually slower (1.5×10^7) than the serial case. If $D = \log(N) \approx 17$, it is almost 4000 times faster.

A.7 Low Diameter Graphs and Pointer Jumping

A.7.1 Low Diameter Graphs

It may be that all the interesting graphs have low diameter, D, such that long paths are not an issue.

Fahlman suggests that one can add nodes to shorten the graph height; certainly this works for long VC (virtual copy) chains. Outstanding questions include:

- does this work for all link types we might need to search?
- can we automatically add these link to keep $D \approx \log(N)$ while keeping the node degree bounded?

A.7.2 Pointer Jumping

Even if the paths are long, we can probably use pointer jumping (*e.g.* [HS86] [Lei92]) so we need to perform only $\log(D)$ marking-passing steps instead of D.

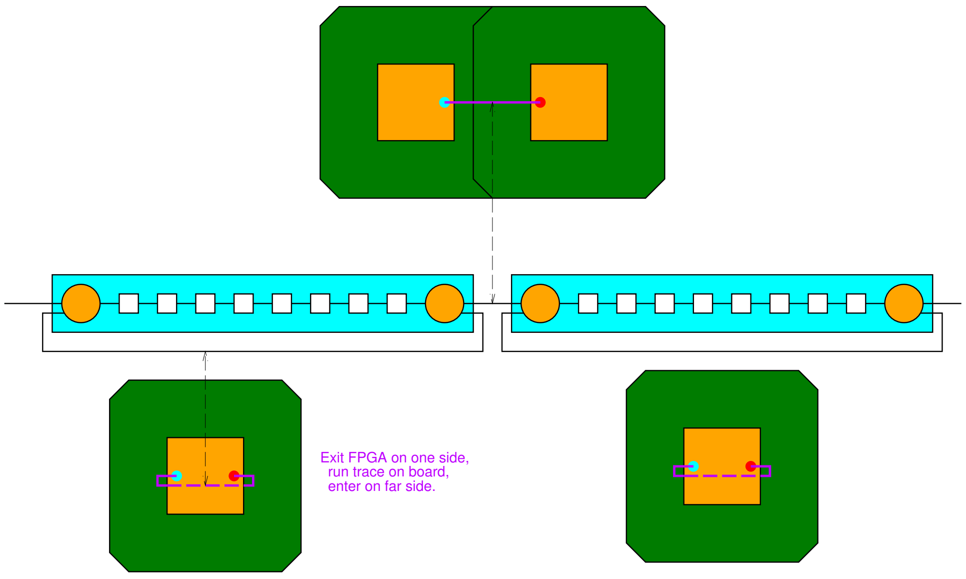
A.7.3 FPGAs and Dishoom Board (2D)

So far, we have collected some anecdotal information:

- At 300 MHz, it takes nine clock cycles to cross a large Spartan 3
- We currently think we can run board-to-board connections (perhaps even crossing an intermediate board) in less than 3 ns (one clock)

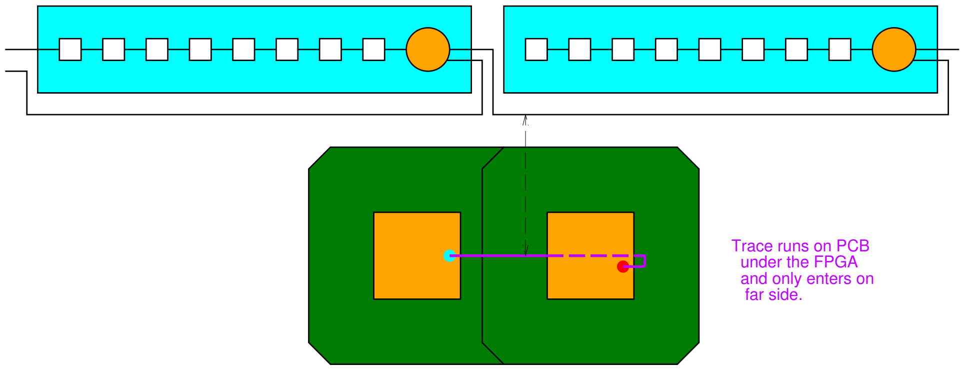
The most straightforward arrangement is to just build a mesh. For simplicity, let's say we put 8×8 PEs on the FPGA. Further, let's assume it takes one clock cycle to cross a PE (maybe that means slowing the clock to 250 MHz for the Spartan), and one extra clock cycle to cross between chips. That means $\alpha = \frac{2 \cdot 9}{8} = \frac{9}{4}$ (the factor of 2 here is for crossing both dimensions).

However, since we should certainly be able to cross the physical distance of the chip on the printed-circuit board in one clock cycle, we can do better if we build some hierarchical wiring. The most conservative (from the PC-board latency standpoint) would be to simply bypass the chip:

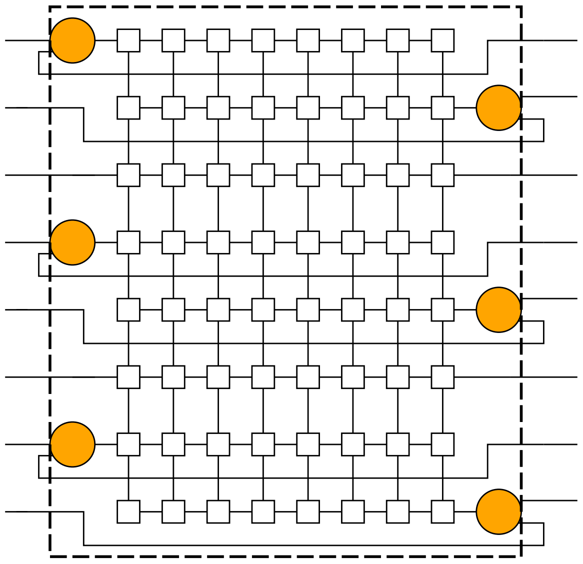


We can now cross chips in three clock cycles, making $\alpha = \frac{2 \cdot 3}{8} = \frac{3}{4}$.

It is not asking much more to be able to cross between boards and enter the far side of the FPGA, as shown in the following figure:

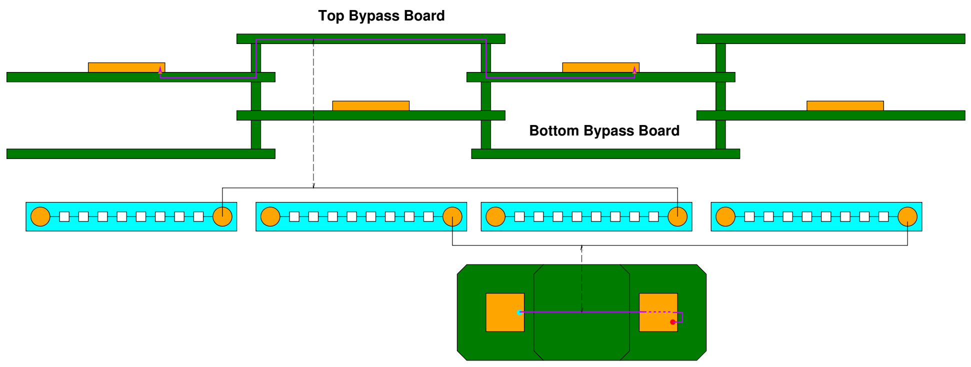


This lets us cross a chip in two clock cycles, bringing $\alpha = \frac{2 \cdot 2}{8} = \frac{1}{2}$. The row shown above is asymmetric in timing and wiring requirements. To compensate, alternate rows in the mesh on the FPGA could have the bypass connection on opposite sides of the chip. Some rows could even have straight-across connections, as shown below:



A similar bypass would be used in both mesh directions (but only the left↔right bypasses are shown in the above figure).

We can add an outer layer of boards to support full board bypass. If we can travel across two board lengths and a pair of connectors in a single clock cycle, this would bring latency down even further:



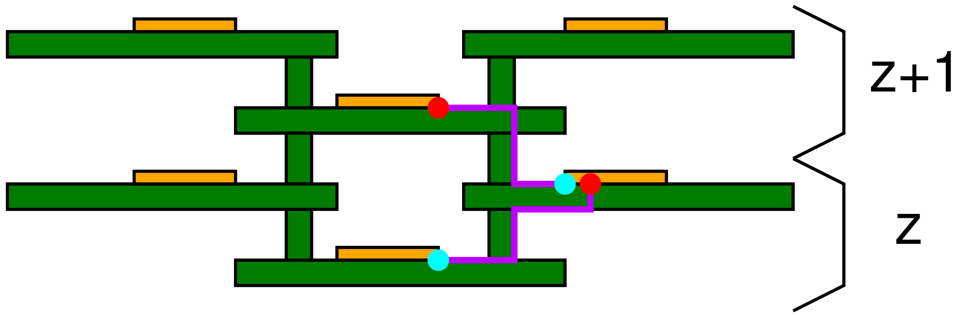
Here we cross two full chips in two clock cycles, so $\alpha = \frac{2 \cdot 2}{16} = 14$. With source-synchronous clocking, we may pay one additional clock cycle crossing the clock boundaries between chips, so a better number might be $\alpha = \frac{3 \cdot 2}{16} = 38$.

A 4096 PE machine would have $T_{lat} = \frac{3}{8} \times 64 = 24$ clocks. At 4 ns, this is a 96 ns cross machine latency.

A.7.4 FPGAs and Dishoom Board (3D)

By continuing to stack Dishoom processing boards, we can go to three-dimensional topologies. Note that a horizontal layer actually exists in two staggered layers in the third (vertical) dimension. Further, when we cross the two boards in a layer, we only change the vertical PE identification by one (unlike in X and Y where we change it by the width or height of the PEs in each FPGA).

The most direct topology starts with nearest-neighbor board connections:



Here we must switch through two links to achieve ΔZ of one. Each board hop requires a chip-to-chip cycle and a switching cycle in the FPGA. Together, this means it takes four clock cycles to cross one horizontal plane in the vertical direction. Consequently:

$$T_{lat} = \alpha_{2d}\sqrt{\frac{N}{N_z}} + \alpha_z\left(N_z - 1\right) \tag{A.32}$$

Here N_z is the number of vertical layers, α_{2d} is the α we have been looking at, which captures distances in each horizontal plane, and α_z is the cycles per plane. In the case above, we noted $\alpha_z = 4$.

To minimize latency, we pick the appropriate N_z based on N. We take the derivative of Equation A.32:

$$\alpha_{2d}\sqrt{N} \times \left(\frac{-1}{2}\right) N_z{}^{\left(\frac{-3}{2}\right)} + \alpha_z = 0 \tag{A.33}$$

$$\sqrt{N} \times N_z{}^{\left(\frac{-3}{2}\right)} - \frac{2\alpha_z}{\alpha_{2d}} = 0 \tag{A.34}$$

$$\sqrt{N} \times N_z{}^{\left(\frac{-3}{2}\right)} = \frac{2\alpha_z}{\alpha_{2d}} \tag{A.35}$$

$$\sqrt{N} \times \frac{\alpha_{2d}}{2\alpha_z} = N_z{}^{\left(\frac{3}{2}\right)} \tag{A.36}$$

$$N \times \left(\frac{\alpha_{2d}}{2\alpha_z}\right)^2 = N_z{}^3 \tag{A.37}$$

$$N_z = \sqrt[3]{N \times \left(\frac{\alpha_{2d}}{2\alpha_z}\right)^2} \tag{A.38}$$

E.g., $N = 2^{14}$, $\alpha_{2d} = \frac{1}{4}$, $\alpha_z = 4$.

$$N_z = \sqrt[3]{2^{14} \times \left(\frac{1/4}{2 \cdot 4}\right)^2} \tag{A.39}$$

$$N_z = \sqrt[3]{2^{14} \times (2^{-5})^2} \tag{A.40}$$

$$N_z = \sqrt[3]{2^{14} \times 2^{-10}} \tag{A.41}$$

$$N_z = \sqrt[3]{16} \tag{A.42}$$

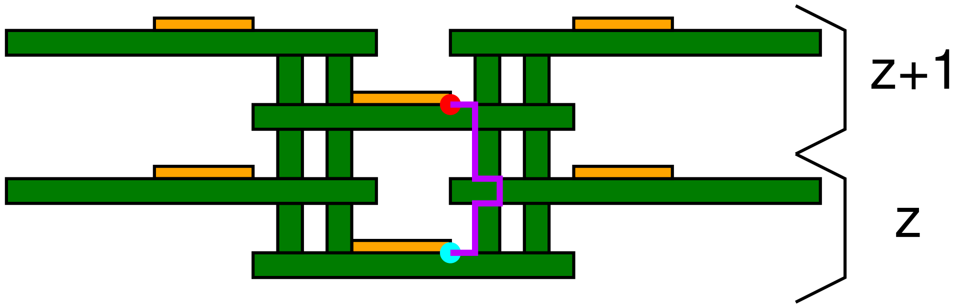
So N_z is between 2 and 3.
In this case, $T_{lat} \approx 27$ for both 2 and 3.

$$T_{lat} = \frac{1}{4}\sqrt{\frac{2^{14}}{2}} + 4 \times 1 \approx 27 \tag{A.43}$$

$$T_{lat} = \frac{1}{4}\sqrt{\frac{2^{14}}{3}} + 4 \times 2 \approx 27 \tag{A.44}$$

This is slightly smaller than $T_{lat} = 32$ for $N_z = 1$.

If we can go through two vertical connectors in one clock cycle, we could add vertical bypass paths that allow us to travel $\Delta Z = 1$ in a single board hop.



This brings α_z down to 2. As shown above, we include a second vertical connector to accommodate the fact that these bypass wires will now be crossing each other, doubling our local wiring requirement.

E.g. $N = 2^{14}$, $\alpha_{2d} = \frac{1}{4}$, $\alpha_z = 2$.

$$N_z = \sqrt[3]{2^{14} \times \left(\frac{1/4}{2 \cdot 2}\right)^2} \tag{A.45}$$

$$N_z = \sqrt[3]{2^{14} \times (2^{-4})^2} \tag{A.46}$$

$$N_z = \sqrt[3]{2^{14} \times 2^{-8}} \tag{A.47}$$

$$N_z = \sqrt[3]{2^6} = 4 \tag{A.48}$$

$$T_{lat} = \frac{1}{4}\sqrt{\frac{2^{14}}{4}} + 2 \times 3 \approx 22 \tag{A.49}$$

A.7.5 Dishoom Bandwidth

Each Dishoom board has eight edge connectors = {N,S,E,W} $\times$ {up,down}. Each of these carries BW_{conn} signals per cycle. The current estimate is $BW_{conn} = 40$.

If $N_z > 1$ we have a bandwidth across a Z-axis bisection:

$$BW_z = 4 \times BW_{conn} \times \frac{N}{N_z} \tag{A.50}$$

Across an X- or Y-axis bisection, we have

$$BW_x = 2 \times BW_{conn} \times N_z \times \sqrt{\frac{N}{N_z}} = 2 \cdot BW_{conn} \times \sqrt{N_z \times N} \tag{A.51}$$

Considering 8×8 planes of Dishoom boards stacked N_z high with $BW_{conn} = 40$, we have

$$BW_z = 4 \times 40 \times 64 = 10240 \tag{A.52}$$

$$BW_x = 2 \times 40 \times 8 \times N_z = 640N_z \tag{A.53}$$

From $BW_{\{x,y,z\}}$ we can calculate a lower bound on T_{load}:

$$T_{load} > \frac{G_{bisect}}{BW_x} \tag{A.54}$$

here G_{bisect} is the bisection of the graph. If we have $N_z > 1$, and the Z-axis bisection bandwidth is larger, the first cut is $\frac{G_{bisect}}{BW_z}$; nonetheless, the second bisection will be only a small constant factor (2^p) smaller and must cross the X-axis (Y-axis) bisection.

For the single plane, 64 PE design with $\alpha = \frac{3}{8}$ considered above, $T_{lat} = 24$. Assuming $BW_{conn} = 40$, how large a bisection can we handle before bandwidth dominates latency?

$$T_{load} > \frac{G_{bisect}}{640} = 24 \tag{A.55}$$

$$G_{bisect} \approx 640 \times 24 = 1536 \tag{A.56}$$

Note, if messages are 32 bits wide, this corresponds to only 48 edges.

For the $N = 2^{14}$, $\alpha_{2d} = \frac{1}{4}$, $\alpha_z = 2$ case above, we computed $N_z = 4$ and $T_{lat} = 22$. Assuming $BW_{conn} = 40$, how large an x-bisection can we handle before bandwidth dominates latency?

$$T_{load} > \frac{G_{bisect}}{640 \times 4} = 22 \tag{A.57}$$

$$G_{bisect} \approx 640 \times 4 \times 22 = 5632 \tag{A.58}$$

If messages are 32 bits wide, this corresponds to only 176 edges.

Note: going to a larger N_z will increase the latency, but can decrease the bandwidth limit. So, if we see substantially larger graph bisections, we may need to pay the extra latency to reduce the bandwidth bottleneck

Appendix B

MATTER Graph Machine Operation Assessment for ConceptNet

Here we begin to sketch the design space for a Graph Machine targeted at supporting the ConceptNet context calculation (spreading activation).

B.1 Node Decomposition

ConceptNet has some very large nodes. If we atomically assigned nodes to PEs (and hence active memories) we would be forced to have very large memories—much larger than a single Virtex block RAM and much larger than the kind of sizes that look promising when we looked at marker-massing in Appendix A. Even aside from memory size, if a large node is atomically assigned to a processing node, it can serve as a serial bottleneck. Consequently, we should consider how we can decompose large nodes into smaller, bounded-degree graph objects. Questions include:

- Should there be a single, fixed-size graph node?
- ...or should there simply be a maximum bound on the execution-level graph node size?
- Should we handle hypergraph links with special edge/fanout nodes? For very high fanouts, should those be decomposed into multiple graph nodes as well?
- Logically, the graph node should be programmed as a single node. What needs to be done so we can efficiently decompose the large node into smaller nodes (*e.g.*, how can we (preferably automatically) figure out the necessary associative transformations for handling data combining)?
- Ultimately, how large should the graph node size bound be?
- How do we place graph nodes on processing nodes (associated memories)?
- How do we efficiently support fanout (fanin)?
- How many bytes per node (base bytes per node + bytes per edge)?
- How do we make the spreading activation score calculation associative?

B.1.1 Graph Node Statistics

Total Edges	0–1	2	3–4	5–8	9–16	17–32
Node Count	41189	106272	72224	49804	25640	9305

Total Edges	33–64	65–128	129–256	257–512	513–1024	1025–2048
Node Count	4771	2664	1584	808	406	170

Total Edges	2049–4096	4097–8192	8193–16384	16385–32768	32769–65536	65537–131072
Node Count	59	12	1	0	0	1

Preliminary cuts suggest that we have a bisection cut around 186,000 edges when the original graph is cut. After thresholding at 128 edges, we get a top cut under 5,000 edges and a second cut around 27,000 edges.

The threshold is a quick way to estimate edge impact of building distributed trees for large fanout nodes.

B.1.2 Graph Object Memory Requirements

$$S_{node} = S_{base} + Edges \times S_{edge} \quad \text{(B.1)}$$
$$S_{edge} = \text{Graph Node Pointer} + \text{Link Type} \quad \text{(B.2)}$$
$$S_{base} = \text{Discount} + \text{Score} \quad \text{(B.3)}$$

With 20 to 30 link types, Link Type will require at least 5b. With $\leq$ 256K nodes, Graph Node Pointer will require at least 18b.

Assuming Discount and Score are 16b each:

$$S_{node} \approx 32 + Edges \times 24 \quad \text{(B.4)}$$

B.1.3 Fanout

Options:

1. No fanout in net (simply a collection of point-to-point links)
2. Net support for fanout
3. Net does not support fanout, but fanout nodes allow efficient message fanout.
   - Placement and topology of fanout nodes based on placement of connections. Perhaps we place the nodes first, then build the fanout tree nodes to minimize communication requirements

B.2 Edge Weighting

Depending on the algorithm, each edge type will be given a different weighting. How do we handle edge weight mapping?

Options:

- Reserve a slot in every graph edge to insert current weight. This is probably prohibitively expensive, almost doubling link area.
- Each processing node has its own translation table, *i.e.*, abstract model is a global table. Implementation pattern is distributed replicas. Update via broadcasts.

1. If number of link types is small, store complete table at every node.
2. If number of link types is large and edges sparsely use link types, store sparse table at every node.

B.3 Sequential Performance Model and Data

We want to decompose the runtime. *E.g.* we have an anecdotal number of 10 to 15ms per query. How does this break down? First, break down into visits and time per visit:

$$T_{alg} = \sum_{v \in V} v.visits \times v.T_{visit} \tag{B.5}$$

We probably want to know how many nodes are visited and the average number of times each node is visited.

$$Total\_Visits = \sum_{v \in V} v.visits \tag{B.6}$$

$$Nodes\_Visited = \sum_{v \in V} ((v.visits > 0)?1 : 0) \tag{B.7}$$

From this we know the average number of visits (updates) per active node:

$$E(visits) = \frac{Total\_Visits}{Nodes\_Visited} \tag{B.8}$$

It will probably be useful to know the maximum and minimum number of visits, as well, *e.g.*,

$$Max\_Visits = \max_{v \in V} (v.visits) \tag{B.9}$$

We will want to break down T_{visit}. One model will be fixed node work plus work per edge:

$$v.T_{visit} = T_{node\_fixed} + v.Edges \times T_{edge} \tag{B.10}$$

We will want to be able to break down each of these costs into operation time and memory time. Perhaps, break down into random memory references (cache misses) and ops/local ops.

$$T_{node\_fixed} = N_{rnd} \times T_{mem\_access} + N_{localops} \tag{B.11}$$

Similarly for each edge:

$$T_{edge} = N_{ernd} \times T_{mem\_access} + N_{elocalops} \tag{B.12}$$

B.3.1 Pragmatic Suggestion for Measurement

Use `ifdef`'s on the code to define various measurements/instrumentation. Run multiple times with different defines to collect all the data.

1. turn off everything and simply capture time for complete job (T_{alg})
2. turn on all event counters (*e.g.*, visits) and turn off all timing counters
3. turn on counter around each node-op only (T_{visit})
4. turn on counter around node-fixed only (T_{node_fixed})
5. turn on counter around per edge processing only (T_{edge})

We may need to do something else (look at assembly, or use some of the other performance counters) to break down memory access times.

B.4 Parallel Execution Model

For conceptual simplicity, we start with the assumption that computation proceeds in steps where we perform on graph update and edge hop per step. This can be relaxed later, but it is easier to think about one set of message hops occurring at once.

At each node, we keep state:

- current-step-max // all this current-step detail is for message digesting
- current-step-min
- current-step-sum
- current-step-count
- my-activity // this is the aggregate (old "score")

Computation proceeds in three phases:

1. Receive messages
   - initialize all current-step variables to 0 (except, perhaps, min)
   - for each message:
     1. current-step-max=max(current-step-max,message-max)

   2. current-step-min=min(current-step-min,message-min)
   3. current-step-sum+=message-sum
   4. current-step-count+=message-count

2. Update node
   - my-activity=f(my-activity,current-step-variables) // assume function is some constant time op
3. Send out updates
   - if ((current-step-count>0) && (current-step-sum>THRESHOLD)) for each outgoing edge: send message with (message-max=current-step-max*edge.weight, message-min=current-step-min*edge.weight, message-sum=current-step-sum*edge.weight, message-count=current-step-count // not weighted
)

B.5 Parallel Performance Model

Key model parameters are

- M_{bits} – message bits
- $E_{in-comp}$ – cycles of computation per input message
- N_{comp} – cycles of computation at node once all step messages arrive // i.e. f above
- $E_{out-comp}$ – cycles of computation per output message

From this, we have

$$T_{gn-comp} = E_{in-comp} \times Max(Inputs) + N_{comp} + E_{out-comp} \times Max(Outputs) \tag{B.13}$$

If $E_{in-comp} = E_{out-comp} = E_{comp}$, then

$$T_{gn-comp} = E_{comp} \times Edges\_per\_node + N_{comp} \tag{B.14}$$

We compose the overall performance much as in Appendix A:

$$\begin{aligned} T_{step} &\approx T_{comp} + T_{comm} && \text{(B.15)} \\ T_{comm} &\approx T_{lat} + T_{load} && \text{(B.16)} \\ T_{comp} &= GraphNodes\_per\_PE \times T_{gn-comp} && \text{(B.17)} \end{aligned}$$

Together:

$$T_{step} \approx T_{comp} + T_{lat} + T_{load} \tag{B.18}$$

For the whole algorithm:

$$T_{alg} \approx T_{bcst} + D \times T_{step} + T_{reduce} \tag{B.19}$$

With

$$T_{bcst} \approx I + T_{lat} \tag{B.20}$$

and

$$T_{reduce} \approx R + T_{lat} \tag{B.21}$$

The assumptions above are that the phases do not overlap. Compute and communicate may be able to overlap, reducing T_{step}. Note, however, that this cannot offer more than a factor of 2 in savings.

We assume here that every node and every edge is active on every cycle. If that is not the case, we can replace $GraphNodes\_per\_PE$ with the maximum number of active graph nodes per PE. Similarly, we may be able to replace the maximum number of input edges with the maximum number of active input edges. By this model, if there is any activity in a graph node, we will send messages out all output nodes. Of course, to get a benefit out of these lower-activity cases, we will need well-balanced graph node clustering so that the maximum number of active graph nodes per PE per step is close to the average number.

B.6 FPGA PE Design Starting Point

Assume that we assign two FPGA Block RAMs per PE. That gives us data 36b wide and 1024b deep. With each edge requiring less than 36 bits, that means we get roughly 1000 edges per PE. If we limit the edges per node to 10, we have about 90 nodes per PE (or if we limit them to 100, we have about 10 nodes per PE). Block RAMs are dual ported, supporting one read and one write per cycle. This suggests a target for $E_{comp} = 1$. That is, on message arrival, we perform one read, compute the update, and then perform one write. We provide a pipelined datapath so that we can receive or initiate one edge message per cycle. The single read and single write are key to making sure we do not have a bottleneck in memory.

The V2-6000 FPGAs have 144 Block RAMs. Consequently, we could put at most 72 such nodes on the FPGA; 64 PEs arranged in an 8×8 grid might be the appropriate target for simplicity. This gives us about 600 4-LUTs per PE. We need to use this logic to both implement the node and provide the interconnect.

Most likely, each PE has an input FIFO built from SRL16s to buffer between the network and the node.

ConceptNet has around 1.6×10^6 edges. Assume that breaking up large nodes gives us 2×10^6 edges. We get roughly 10^3 edges per PE, so we will need at least 2000 PEs to hold ConceptNet. With 64 PEs per FPGA, a minimum of 32 FPGAs is needed to hold the entire ConceptNet; 64 FPGAs is probably a more comfortable number to allow for uneven packing of the graph nodes into the PE node memory. Examples from Appendix A suggest that we can go to larger machines and reduce the runtime.

Consider using 8×8 planes of Dishoom boards as the base and stacking N_z of those high. Assume $E_{comp} = 1$ and $N_{comp} = 0$ (for simplicity, assuming it will be dominated by E_{comp}). Take the Dishoom bandwidth and latency from Appendix A. Assume $G_{bisect} = 3 \times 10^4$ and each node message is 32b wide.

$$\begin{aligned} T_{comp} &\approx \frac{2 \times 10^6}{64 \times 64 N_z} && \text{(B.22)} \\ T_{load} &\approx \frac{3 \times 10^4 \times 32}{640 \times N_z} && \text{(B.23)} \\ T_{lat} &\approx \left(\frac{3}{4}\right) 64 + 4\left(N_z - 1\right) && \text{(B.24)} \end{aligned}$$

Collapsing more constants:

$$\begin{aligned} T_{comp} &\approx \frac{5 \times 10^2}{N_z} && \text{(B.25)} \\ T_{load} &\approx \frac{1.5 \times 10^3}{N_z} && \text{(B.26)} \\ T_{lat} &\approx 48 + 4\left(N_z - 1\right) && \text{(B.27)} \end{aligned}$$

This gives us

N_z	N_{FPGA}	T_{comp}	T_{load}	T_{lat}	T_{step}
1	64	500	1500	48	2048
2	128	250	750	52	1052
4	256	125	375	60	560
8	512	63	188	76	327

Running at 200 MHz, this is 1.5→10 μs per step. Assuming 8 steps, this is 12 to 80 μs for the application (maybe say 10 to 100μs).

The sequential version runs in 10 ms, so we can estimate a performance improvement of two to three orders of magnitude.